POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

On the Performance of Hypervisor-assisted Memory Monitoring for Code Unpacking Detection

Tesi di Laurea Magistrale in
Computer Science and Engineering - Ingegneria Informatica

Author: **Alessandro Mileto**

Student ID: 993653
Advisor: Prof. Mario Polino
Co-advisors: Alessandro Bertani, Prof. Michele Carminati
Academic Year: 2022-23

Abstract

In the ever-evolving cybersecurity landscape, malware continues to pose significant threats, evading and subverting analysis and detection tools with techniques like code packing. Run-time packers compress and/or encrypt malicious code to obfuscate it, but packed code must be decompressed/decrypted in memory before execution. Tracing memory accesses can help to detect unpacking attempts by malicious programs to secure systems.

Virtual machines have long been used for malware detection and analysis since the hypervisor sits at a privileged position and has broad visibility on the system, allowing it to intercept memory accesses and detect suspicious activity. Researchers in both industry and academia proposed various monitoring solutions, but their performance overhead has not been thoroughly studied yet. This thesis aims to address this issue, restricting the view to the slowdown resulting from real-time code unpacking detection.

We study two solutions, Drakvuf and HVMI. We show that in the worst case, these tools introduce a slowdown factor of more than `2000x` on memory accesses but, when monitoring real-world programs, performance penalties are at most `3.80x` for Drakvuf and `1.11x` for HVMI. Despite HVMI being efficient and potentially suitable for real-time monitoring, we identify a corner case not correctly handled by it.

We also explore the potential of the hypervisor-based debugger HyperDbg. We retrofit it as a monitor for code unpacking detection and show that, although HyperDbg-based memory tracing is less invasive than HVMI and Drakvuf from a performance standpoint, the tool is not a viable option for efficient memory analysis.

The contributions of this thesis are an analysis of the performance overhead on memory accesses in the presence of memory hooks, a characterization of the impact of hypervisor-based monitoring for code unpacking detection, a comparison between two different approaches for memory hooking and execution tracing, and the definition of guidelines for the implementation of hypervisor-based monitoring for real-time code unpacking detection.

Keywords: Hypervisor, memory, monitoring, unpacking

Abstract in lingua italiana

Nel costante evolversi del panorama della sicurezza informatica, i malware rappresentano una minaccia importante, eludendo gli strumenti di analisi e rilevamento con tecniche come il packing del codice. I packer sono programmi che comprimono e crittografano il codice maligno per offuscarlo, ma il codice "packato" deve essere decompresso in memoria prima di essere eseguito. Tracciare gli accessi alla memoria può aiutare a individuare tentativi di "unpacking" da parte di malware e contribuire alla sicurezza dei sistemi.

Le macchine virtuali sono da tempo utilizzate per la rilevazione e l'analisi del malware, grazie alla posizione privilegiata del hypervisor e alla sua ampia visibilità sul sistema. I ricercatori in accademia e industria hanno proposto varie soluzioni di monitoring ma il loro impatto sulle performance non è stato adeguatamente studiato. In questa tesi puntiamo a colmare questa lacuna restringendo il campo di studio alla rilevazione in tempo reale di "unpacking" di codice.

Studiamo due soluzioni, Drakvuf e HVMI. Nei nostri esperimenti mostriamo che, nel caso peggiore, il fattore di rallentamento sugli accessi in memoria introdotto supera `2000x`, ma non per i programmi reali dove si attesta a `3.80x` per Drakvuf e `1.11x` per HVMI. Anche se HVMI permette di monitorare in tempo reale, noi identifichiamo un caso particolare in cui si dimostra inaffidabile.

La tesi esplora anche le potenzialità del debugger basato su hypervisor HyperDbg. Lo adattiamo al task del monitoraggio della memoria per il rilevamento di "unpacking" e dimostriamo che, sebbene sia meno invasivo rispetto a HVMI e Drakvuf dal punto di vista delle prestazioni, lo strumento non è un'opzione valida per un'analisi efficiente.

I contributi alla ricerca di questa tesi includono un'analisi dell'overhead prestazionale sugli accessi alla memoria dovuto al monitoring, la caratterizzazione dell'impatto del monitoraggio basato su hypervisor per la rilevazione dell'unpacking di codice sulle prestazioni delle applicazioni reali, il confronto tra due diversi approcci per risolvere il problema, e la definizione di linee guida per l'implementazione di soluzioni efficient.

Parole chiave: Hypervisor, memoria, monitoraggio, unpacking

Contents

1 | Introduction

Malware can be generically defined as any piece of software intentionally attempting to execute malicious code on a target system. With approximately 1 million malware files created every day and 6 trillion dollars of damage to the world economy in 2021, malware is a serious challenge to cope with [13].

Malicious payloads may vary in shape, size, and privilege level at which they run. Malicious samples often implement evasive behavior to avoid being detected by antivirus solutions and analyzed using ad-hoc frameworks [42]. Analysis and monitoring solutions subversion is easy if malware runs at the same privilege level: in this case, the malware has control over their view of the system.

Some families of evasive malware may exhibit packing, an obfuscation technique based on code compression and/or encryption to evade malware analysis and detection techniques. The one thing that packed malware samples have in common is that payloads have to be written into memory before being executed [52]. Capturing potential code unpacking attempts can help to detect packed malware inside a system. One possible strategy to do so is tracking the memory accesses performed by processes.

Virtualizing means creating a duplicate of a physical machine for an operating system, a **virtual machine**, controlled by a piece of software called **hypervisor** [26]. Virtualization-based protection measures have been proposed over the years to isolate workloads, confine malicious payloads in a virtual environment, and potentially counteract evasive malware behavior for analysis purposes. Some use cases even include the detection of malicious software using antiviruses, which run at the hypervisor level and track state changes in the virtual machine in response to memory accesses, like the execution of suspicious memory regions. They are based on the assumption that if malware runs in a less privileged context than the hypervisor, it cannot hide from monitoring solutions, since they have a complete view of the monitored target [14].

The main problem hypervisor-based monitoring solutions solve is the so-called *semantic gap*, i.e., the disparity between the low-level data obtained by inspecting the state of the virtual machine and the high-level information required by analysis tools. The approach

to filling such a gap is called **virtual machine introspection** and it leverages knowledge of the internals of the monitored operating system coupled with the state of virtualized hardware for state reconstruction.

To track changes in a virtual machine and dynamically update the representation of its state, introspection libraries leverage the capabilities of modern hypervisors to place hooks inside the virtualized system. A **hook** is a mechanism to intercept specific events happening inside the virtual machine and control their execution. Memory hooks are often implemented using extended memory management facilities of modern processors. Researchers from both industry and academia proposed a plethora of introspection solutions implementing various approaches for hooking and state reconstruction leveraging such facilities. For our analysis, we picked the academic projects Drakvuf and HyperDbg and the industrial one HVMI.

Drakvuf is a sandbox for hypervisor-based malware analysis and reverse engineering that runs on the popular Xen hypervisor [34]. It is based on LibVMI [44], a state-of-the-art virtual machine introspection library leveraging kernel debugging data to build a representation of the state of the virtual machine. Drakvuf uses it to support monitoring solutions implemented as plugins.

HVMI is an operating system hardening framework developed by BitDefender to counteract cybersecurity menaces at the hypervisor level. It leverages the Introcore VMI library to face the semantic gap, which extracts patterns from the kernel of the monitored operating system to infer its memory layout at runtime, reconstruct its internal state, and inject in-kernel hypervisor-protected code.

Drakvuf hooks the memory of a virtual machine instantiating multiple views over its physical memory, a technique also known as **altp2m**. HVMI, instead, **alters** the permissions of memory pages at the hypervisor level, and redirects the control flow of the guest kernel using **detours**. A detour is an in-guest code snippet protected from tampering and detection using the hypervisor. These mechanisms for coping with the semantic gap and execution control enable the analysis of the execution of a virtual machine, potentially helping to detect malware. However, the state of the art misses an in-depth analysis of the performance overhead introduced by existing solutions. [35].

In this thesis, we focus on the task of **code unpacking detection** at the hypervisor level through memory access tracing at the memory page granularity. More precisely, we aim to investigate its **performance overhead** on the execution of the processes in a virtual machine. We examine strategies for hypervisor-based monitoring based on Drakvuf and HVMI and study their impact on the execution of real-world programs.

As a starting point, we wish to ask ourselves the following questions:

- **What is the overhead on memory operations deriving from memory hooking?**
- **What is its performance impact on the execution of real-world programs and on their usability?**
- **Is real-time unpacking detection at the hypervisor level suitable for commodity systems?**

To answer them, we test the two chosen solutions against a synthetic benchmark for memory hooking, discovering that the slowdown factor on memory accesses amounts to **2120x** and **2212x** for Drakvuf and HVMI respectively. Nonetheless, we show that this overhead is not experienced when monitoring real-world benign programs: in this case, the performance penalty factor is at worst **3.80x** for Drakvuf and **1.10x** for HVMI.

We conclude that HVMI is more efficient thanks to the way it fills the *semantic gap* and that this approach to hypervisor-based monitoring may be used for **real-time** unpacking detection. At last, we evidentiate a corner case not correctly handled by HVMI and provide insights on the main **limitation** of its approach to code unpacking detection, that Drakvuf has not.

As a secondary research question, we take into consideration the HyperDbg hypervisor-based debugger, designed for supporting malware reverse engineering. Its memory hooking capabilities make it a potential candidate for memory monitoring for unpacking detection, and we try to retrofit it as a monitor using its powerful hypervisor-level scripting engine. We write a debugger program for memory monitoring and investigate the functional equivalence between the proposed script and the monitoring strategy implemented in HVMI and Drakvuf, highlighting the performance degradation of virtual machine memory operations. We discover that HyperDbg-based tracing is less invasive than HVMI and Drakvuf from a performance standpoint with its **771x** slowdown factor on memory operations, but the tool is not a viable option for scalable memory analysis. To prove this last part, we retrofitted the benchmark program written for assessing hooking penalty on HVMI and Drakvuf, to study the behavior of the debugger in response to the progressive extension in length of the monitored region. We show that the more instructions are fetched from the location of interest, the higher the slowdown on the guest, quickly reaching a factor of **2048x** for minimal benchmark modifications.

The contributions of this thesis to the state of the art are the following:

- An analysis of the performance overhead **on memory accesses** in the presence of hypervisor-level memory hooks
- A characterization of the impact of hypervisor-based monitoring for code unpacking detection on the performance of **real-world applications**
- A comparison between **two** different approaches for memory hooking and solving the semantic gap problem
- The definition of **guidelines** for the implementation of hypervisor-based monitoring solutions for real-time code unpacking detection.

2 | Motivation and background

This chapter describes the core research problem studied in the present thesis. We present the most important technical concepts to provide some background knowledge on virtualization technologies. Then, hypervisors and hypervisor-based monitoring are introduced to pave the way for the discussion of real-time malware and, more specifically, code unpacking detection.

2.1. Technical background

This section describes the basic notions to understand the core research challenges of this thesis.

2.1.1. Virtualization

As defined by Popek and Goldberg [45], virtualization is about creating a duplicate of a physical machine, i.e. a virtual machine, with some specific requirements that entail fidelity (the behavior of software in the virtualized environment has to match its behavior in real hardware), efficiency (limited performance overhead on the virtualized software execution) and isolation (access to resources by virtual machines has to be strictly regulated). The component in charge of virtual environment creation and management is called hypervisor (or virtual machine monitor). It multiplexes physical hardware, exposes it to virtualized software, arbitrates access to system resources, and enforces separation among the co-hosted virtual machines in accordance with the above requirements. A first taxonomy of virtualization solutions can be proposed distinguishing Process Virtual Machines and System Virtual Machines. Process virtual machines are either based on the concept of resource multiplexing contemplated in commodity OS es (multi-programming) or on binary translation/emulation techniques used to run programs written for a given instruction set architecture on a machine based on a different one. System virtual machines provide an operating system instance with a virtual environment which is a resource-constrained duplicate of the hardware of the physical machine on which the virtualized system runs [20].

System hypervisors fall into two categories [14]:

- Type 1: the hypervisor runs on bare metal and it has complete control over the computer hardware. Common examples are Xen, Napoca by BitDefender, ProxmOS, and Microsoft HyperV.
- Type 2: the hypervisor runs on top of an operating system instance (Host), integrated in a host component which mediates access to the hardware by the virtual machines (Guests) interfacing with the host kernel. Common examples are VirtualBox, VMWare Workstation, QEMU, and KVM.

Modern operating systems and computers revolve around the concept of privilege separation. The operating system runs at the kernel privilege level, while user-mode applications are less privileged and cannot directly control hardware and system internals. Modern computer architectures enforce privileged systems component separation from unprivileged ones adopting a hardware-based protection model. Operations only available at certain privilege levels are said to be privileged. On the contrary, unprivileged operations are available at every level and, according to how instruction set architectures are designed, they usually cannot strongly affect the overall system state. According to the protection model, if the software running at an insufficient privilege level tries to execute privileged instructions, a protection fault is generated by the hardware [5]. Implementing virtualization on some hardware architectures may not be straightforward due to compatibility issues. In some cases, unprivileged operations may reveal the presence of the hypervisor to the virtual machine or behave differently in a virtualized context. Popek and Goldberg studied this phenomenon and created a Theorem stating that in virtualizable hardware architectures instructions whose behavior is sensitive to virtualization are a subset of privileged operations [45]. The design of modern hardware for virtualization has been strongly influenced by this theorem.

Hypervisors may share the same privilege level as the host operating system kernel, as in the case of type 1 hypervisors. Alternatively, they can sit behind the kernel of the host (Type 2), possibly de-privileging it. Historically, the first virtualization systems revolved around the Software-based virtualization paradigm [20]. In software-based virtualization privileged instructions are run in de-privileged mode, the hypervisor catches the faults stemming from privileged instructions execution at an insufficient privilege level and handles them using emulation (Trap and Emulate). Some instructions that cause compatibility issues when run in a virtual machine are dynamically translated by the hypervisor at guest run-time to virtualization-compatible ones. Later, Hardware-based virtualization emerged to ease the adoption of virtualization and to solve some problems

associated with the Software-based counterpart[53]. Hardware-based virtualization leverages hardware extensions to introduce new operation modes to distinguish between the Host the hypervisor, and the Guests which are complementary and orthogonal to classical privilege separation abstractions of commodity Operating Systems (kernel and userland) and define new privilege levels.
One substantial difference between software and hardware-based virtualization is in the way memory is virtualized. Guest operating systems are not given full control of physical memory as this may interfere with the execution of the VMM and violate the isolation requirement stated above [20]. As explained by Zhang in [59], in software-based virtualization, the hypervisors virtualize memory leveraging Shadow Page Tables, data structures defined by the hypervisor. Every page table modification occurring in the guest traps in the VMM and it adjusts the entries of the shadow page tables accordingly. This way, the guest is given the illusion to be managing physical memory directly while, instead, it is not. Memory accesses happen at native speed, but trapping in the hypervisor for every little change is costly.

On the contrary, in hardware-assisted virtualization, hardware extensions implement Second Layer Address Translation (SLAT). SLAT is based on a physical extension of paging hardware which introduces a second level of translation from guest physical addresses to host machine addresses. This enables the creation of simpler hypervisors, since the hardware supports memory paging extension and traps in the hypervisor for managing memory are substantially reduced [5]. Nonetheless, it introduces slowdowns in memory accesses due to the extra indirection [27]. As of today, hardware-based virtualization has become the de facto standard in that in modern architectures since guest execution can happen at near-native speed factoring out continuous traps in the hypervisor and exploiting efficient hardware compatibility layers [5, 53].

The most common use cases of virtualization include workload consolidation to counteract hardware under-utilization in data centers, isolation of computational workloads hosted on the same infrastructure, and workload migration implemented encapsulating computation inside the abstraction of a virtual machine instance[53], but it also has security-related use cases like sand-boxing for reverse-engineering [34], automated vulnerability discovery in kernel code [51] and OS debugging [7, 31] and monitoring [25].

2.1.2. Intel x86: internals and VT-x

Intel x86 architecture implements the protection model referenced in section 2.1.1 passing through the ring abstraction. Each ring has its associated privilege level and certain

operations involving the hardware and the handling of low-level system routines are only available to software running at some privilege rings. The x86 architecture defines four privilege rings, from the most privileged ring 0 to the less privileged ring 3, but in practice only ring 3 and ring 0 are used to implement modern operating systems, for user and kernel space respectively [5] [58].

The situation is different when modern hardware extensions are taken into consideration. Intel VT-x is a set of hardware extensions for Intel x86-64 processors to ease virtualization implementation on modern systems and to solve some of the software virtualization issues. VT-x defines two modes of operations, which are different running contexts used for privilege separation: VMX root mode for the hypervisor and the host system and VMX non-root mode for guest virtual machines. Both modes integrate the classical four privilege levels of x86 security architecture [53] but VMX root mode introduces ring -1, the highest privilege ring, where the hypervisor runs [58].

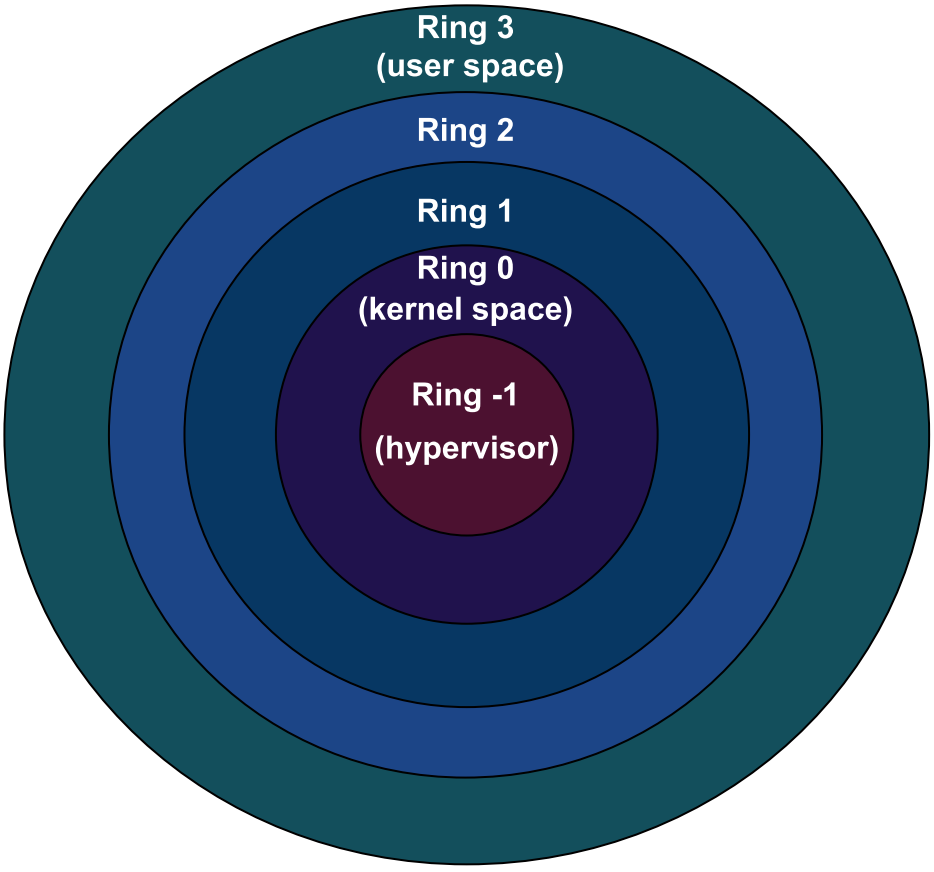

Figure 2.1: Simplified view of protection rings in x86 architecture.

The execution of the guest happens in VMX non-root mode according to the configuration established by the VMM through the VMCS data structure (Virtual Machine Control

Structure) [53]. There is one such structure for each virtual machine. Before the creation of a VM, the VMCS is instantiated and populated inside a hypervisor-controlled memory region, whose handle is placed in a protected CPU-controlled location [5]. Upon setup, the lifecycle of the guest is described in Figure 2.2.

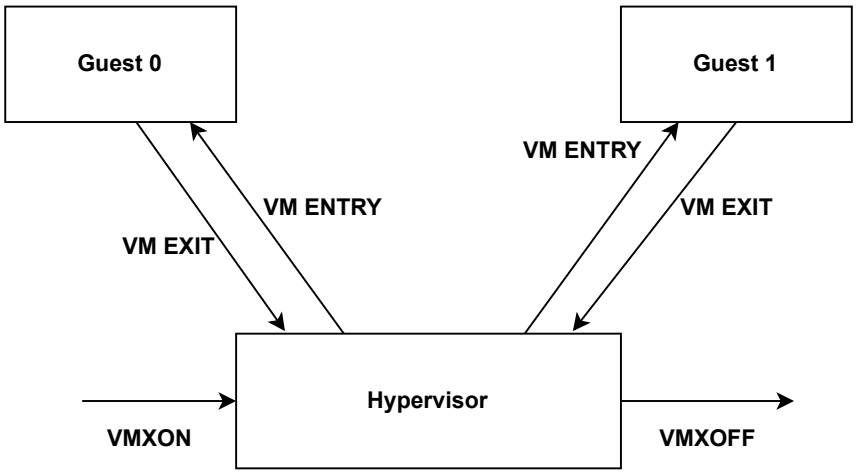

Figure 2.2: Interaction between virtual machines and hypervisor.

The VM-Exit operation is a context-switch from the guest virtual machine to the hypervisor, i.e. a switch from VMX non-root mode to VMX-root mode. Upon a suitable VMCS setting, VM-Exits can be programmed to happen in response to a vast variety of guest events. Upon the context switch from VMX non-root mode to VMX root mode, the state of the guest is saved inside the guest-dedicated area of the VMCS. Similarly, when guest execution restarts host state is saved into a host-dedicated area. This is how VT-x implements state preservation across guest-to-host and host-to-guest transitions[5] [53]. As reported in Intel Software Development Manual [5], Intel VT-x extends the x86 instruction set with instructions used in VMX root mode to configure and manage virtual machines.

The instructions in Table 2.1 can be used to manipulate the VMCS either at guest creation time or at runtime. The hypervisor leverages them to dynamically change the view of the hardware provided to the guest, to enforce specific policies over its operations, to capture particular events such as read/write operations to particular memory locations or registers, for interrupts configuration and handling, virtual I/O management, etc [5].

VMX introduces new cross-world communications primitives summarized in Table 2.2.

The implementation of SLAT in Intel VT-x is known as Extended Page Tables (EPT) [5]. Upon allocation by the hypervisor, a reference to the EPT data structures inside VMM

Name	Description
VMREAD	This instruction reads a VMCS component and stores it into a destination operand.
VMWRITE	Writes the content of one input operand to a location in VMCS.
VMLAUNCH	Launches a virtual machine, i.e., a VM entry occurs, giving control to the VM.
VMRESUME	Resumes a virtual machine leading to a VMentry.
VMXOFF	Forces the processor to leave VMX operation.
VMXON	Causes a logical processor to enter VMX root mode and use the memory referenced by the input to support virtualization.

Table 2.1: Some of the VMX extensions to the x86 ISA (from Intel SDM).

Name	Description
VMCALL	Software in VMX non-root operation calls the hypervisor for service. A VMexit occurs, and control passes to the VMM .
VMFUNC	Software in VMX non-root mode invokes a VMfunction, a processor functionality enabled and configured by software in VMX root mode with no VMexit.

Table 2.2: VMX Instructions: VMCALL and VMFUNC, from Intel SDM.

-controlled memory is placed inside an ad hoc area of the VMCS named the EPTP. It is a pointer to the current Extended Page Table in use by a guest operating system. Such a pointer is modifiable during the guest runtime to change its actual view on paged memory [5]. VT-x compatible hardware can handle up to 512 EPT views existing at once in a given system. Modern hypervisors exploit such capability to maintain different page table views (EPT views) for each guest during the execution of the virtual machine and for switching between them to alter the memory view of the guest [33]. VT-x also introduces extensions to the traditional memory permissions system of Intel processors to introduce finer permissions granularity with execute-only pages and sub-page permissions to enforce different permissions settings for different portions of a given page. Extended page permissions violations usually trap in the hypervisor. Such events fall in the broader category of EPT violations, as described in Intel SDM [5]. As an alternative, such violations can be efficiently converted into exceptions raising an interrupt in the guest leveraging a mechanism called Virtualization Exception to limit VM-Exits [5].

Modern Intel processors also implement debugging extensions for virtualization. Upon configuration, software and hardware breakpoints in the guest can trap in the hypervisor. This feature enables guest debugging and tracing. In addition, the Monitor Trap Flag is a special flag inside the VMCS of a Virtual Machine. It is the VT-x equivalent of the Trap

Flag bit inside the RFLAGS register. When set to 1, Trap Flag makes each instruction fetch raise a breakpoint exception in the process that is running the instruction. On the contrary, the Monitor Trap Flag does not generate an exception that traps into the operating system kernel since the guest directly traps into the hypervisor.

Hypervisor-based monitoring solutions make large use of this feature for guest instruction execution single-stepping and fine-grained debugging. System calls can either be implemented using interrupts (legacy mode) or exploiting ad hoc hardware extensions introduced for efficiency. In the first case, the system call is an interrupt whose handler address is stored in the interrupt descriptor table of Intel processors (IDT). Monitoring solutions can either modify the IDT entry or directly patch the guest system call handling routine, which is a kernel function in charge of handling invocations of the operating system by user mode processes. Hardware-assisted System calls tracing, instead, is usually realized either exploiting model-specific registers. SYSENTER_EIP_MSR contains the address of the operating system system call handler. If its value is modified, the invocations of SYSCALL instruction may result in custom behavior. IA32_EFER MSR can be used to either disable or enable the SYSCALL/SYSRET and the SYSENTER/SYSEXIT instructions. If disabled, when the fetch of one of those instructions happens, the processor raises an Undefined Opcode exception which can be set up to trap in the hypervisor [5, 22, 31, 34].

2.1.3. Modern hypervisors

This section presents the capabilities and facilities integrated into hypervisors that abide by the requirements for VMM s to fully support Virtual Machine Introspection described in the paper *A Virtual Machine Introspection Based Architecture for Intrusion Detection* by Garfinkel at al. [25]. Such a hypervisor is said to be introspection-ready. Apart from isolation, a general requirement of virtualization described in Section 2.1.1, introspection-ready hypervisors must satisfy the Inspection and Interposition requirements to be able to extract low-level information from the running guest and to intercept some actions of interest in it. Starting from these specifications, modern introspection-ready hypervisors were designed and then introspection-ready hardware (like Intel VT-x compatible processors, as described in Section 2.1.4) was introduced to ease introspection adoption. Introspection-ready hypervisors expose APIs to the host and, sometimes also to the guest itself, for manipulating and querying low-level information about the execution state of a virtual machine. APIs can be seen as just wrappers over assembly routines manipulating the VMCS and related control data structures in the hypervisor which possibly implement supplementary actions to perform accounting and/or call hypervisor-specific routines [11].

Such APIs can be reunited into families displayed in the following:

- EPT view creation, modification, switching, and destruction APIs
- Memory permissions modification APIs
- Event channels-based message-oriented communication APIs
- Register reading and writing APIs
- Register operations tracking (e.g. Writes to CR3, Control MSRs, TSC)
- Instruction emulation APIs
- Guest debugging APIs (e.g. Enable single-stepping setting MTF to 1)
- Guest memory mapping, remapping, and extension APIs
- VM-Exit conditions manipulation, callbacks definition, and registration APIs

The advanced internal communication infrastructure enables user or kernel mode libraries to perform event-based monitoring of the guest analyzing system calls invocation, interrupts, and memory accesses. At a very high level of abstraction, monitoring components invoke the hypervisor to register for notification of certain events generated by the guest and possibly define custom callbacks in response to them. Callbacks placement is an important design choice for hypervisor-based monitoring solutions. Some libraries install them in the userspace of the host, while others move them inside the hypervisor. When running in user space inside the host, events have to be forwarded from the guest to the hypervisor and then to the host while responses travel in the opposite direction. If a callback is installed in the hypervisor, the extra communication step from the hypervisor to the host and back is not needed. In this latter context, callbacks run at the same privilege level as the hypervisor, and hypercalls from the host to the hypervisor are not needed. Depending on their placement, callback dispatch and execution may be faster or slower.

Whenever event-based monitoring is set up, upon the occurrence of an event of interest a vm-exit may happen. In such case, the state of the guest is saved in the VMCS, the guest is stopped by the hypervisor and the VMM may either handle the guest exit itself or forward the event to the event channel associated with the suspended virtual machine. The registered libraries, if any, parse messages from the event channel linked to the virtual machine that generated the event leading to a trap in the hypervisor [11] to figure out the cause of the exit. The custom callbacks, independently of their placement, implement actions that reconstruct, analyze, and possibly alter guest execution state before letting

the hypervisor restart it. To cope with the astonishing variability of operating systems, a supplementary software layer is needed for deep state reconstruction and manipulation since such hypervisors have a limited understanding of in-guest dynamics[25]. This is the point where Virtual Machine Introspection comes into play.

2.1.4. Virtual Machine Introspection

As defined by Garfinkel et al. in [25], Virtual Machine Introspection is the task of inspecting a virtualized operating system from the outside aiming at analyzing the events happening in it. VMI was born out of the intuition that Intrusion Detection Systems can leverage the hypervisor to fully intercept the interaction among the hardware and guest virtual machines for security and protection purposes leveraging the isolation granted by the different privilege level at which the VMM sits[25]. The main problem VMI has to solve is called the **semantic gap**. The semantic gap problem refers to the disparity between the low-level data obtained by inspecting a virtual machine's state and the high-level information required by security or analysis tools [57]. It can also be described as the extraction of high-level semantic information (e.g. process names, opened files, etc.) from the operating system's raw memory and virtualized hardware states [25].

Many approaches to solving such problems have been implemented and LibVMI is among these. It evolved from the legacy XenAccess library and exposes APIs to perform read and write operations on guest operating system memory and registers [44]. It is integrated with many famous VMM s such as Xen, KVM, and Bareflank and also works on memory snapshots on both Linux and Windows. It runs in a separate virtual machine, which is generally identified with the host, than the one being introspected and can be integrated with memory forensics tools like Volatility [44] and malware analysis libraries [34]. It solves the problem of the semantic gap leveraging OS -specific information coming from the debug information of the guest operating system kernel [6]. Upon finding the base memory address of the kernel inside the memory of a virtual machine by linear search for specific byte signatures, the kernel structures and function offsets extracted from the debugging information are used to create an internal representation of the memory layout of the VM. Invoking the APIs of the library, such representation is used to read and write specific memory locations identified by the associated symbols [57].

Drakvuf is an extensible binary sandboxing library based on LibVMI leveraging Virtual Machine Introspection for malware analysis and reverse engineering. Several plugins can be created or are already integrated into the library to implement monitoring and analysis-specific actions such as tracking syscall invocation, synchronizing with the guest

scheduler, and inspecting its structures to detect hidden processes, etc [34].
Introcore is an introspection library developed by BitDefender compatible with Linux, Windows, and CentOS . It runs on top of KVM, Xen, and Napoca, a custom hypervisor written by BitDefender for making operating system hardening efficient and easing Introcore integration at the hypervisor level. Introcore can both run at the user level in a host system which manages and introspects several virtual machines (the guests) or can be built and integrated at the hypervisor level thanks to its highly modular and flexible architecture. The library introspects guest operating systems using kernel information extracted programmatically from the guest operating system kernel binary. It extracts system call signatures and displacements from the kernel base, kernel internal routines signatures, and user and kernel space structures from the kernel binary, and, integrating debugging information deriving from kernel debugging symbols, it builds an overlay file to support introspection. Given the virtualized kernel run-time data, the library builds an internal representation of guest memory layout which enables memory and register hooks, deep process introspection to closely follow process creation, destruction and detect hijacking, track dynamically linked libraries invocation, and more. The library has been integrated into the HVMI framework which implements customizable policies for operating system hardening and malware protection.

VMI has many use cases apart from protection and security enforcement, such as debugging [31, 46], malware analysis and reverse engineering [34].
Given an introspection-ready hypervisor and a VMI solution to solve the semantic gap problem, hooks can be placed in the guest to closely follow specific events (e.g. instruction fetches in a particular memory location, writes to kernel data structures, etc.) by suitably setting the control structures associated to virtual machine execution at the hardware and hypervisor level [28].

2.1.5. VMI debuggers

One of the most interesting and recently growing use cases of Virtual Machine Introspection is high-performance, high-privileged debugging of guest virtual machines bypassing kernel and user space debugging facilities provided by the guest kernel [31]. Historically, several attempts have been made to implement a debugger leveraging the hypervisor to debug and instrument a virtual machine without invoking kernel debugging APIs [32, 46]. Hypervisor-based debuggers use VMI to reconstruct the state of guest processes and exploit introspection-ready hardware features to follow guest execution across user and kernel modes seamlessly. Hypervisor debuggers extensively rely on hardware extensions and sit in a privileged position. This makes them potentially less invasive than conventional

APIs-based debuggers and suitable for evasive malware reverse engineering, in-depth performance evaluation, and optimization alongside other forensics use cases [31].

HyperDbg is a research project explicitly created for advanced malware reverse engineering and analysis. It leverages VMI on top of extended introspection-ready hardware features like the ones described in section 2.1.2. The debugger is also a hypervisor, capable of virtualizing an already running guest OS and scriptable using a command line interface. The scripting engine is used to implement classic debugging flows and actions (breakpoints and watchpoints management, disassembly, etc.). What makes HyperDbg special are the scriptable, advanced, powerful, introspection-based monitoring actions. They leverage EPT-based stealthy hooks and detours to evade hypervisor-detection attempts put in place by malware[31] and efficiently track its execution to collect data and reverse-engineer it. Despite EPT-based hooking being already available on most commodity systems they neither virtualize an already running operating system instance nor provide the user with an easy-to-use and expressive script engine leveraging advanced virtualization extensions for efficient debugging [11, 31].

2.1.6. Malware

Malware can be generically defined as any piece of software intentionally attempting to execute malicious code on a target system. Malicious payloads may vary in shape, size, and privilege level at which they run [42]. Malware behavior has been changing rapidly in recent years and its complexity has been growing very fast together with the damage to computer users due to malware infections [13, 43]. The problem of detecting a malicious program, i.e. telling apart malware from benign software, is believed to be NP-hard [13], so, in practice, malware detector programs all use heuristics to try to detect malicious software. Aslan et al. [13] describe the various phases of the detection process: analysis, feature extraction, and the use of detection algorithms over the extracted features. Analysis techniques fall into two categories: static and dynamic. The former consists of extracting features from one program without executing it, while the latter is about observing malware execution to capture its behavior. However, new-generation malware employs sophisticated obfuscation techniques to evade detection systems and hinder static and dynamic analysis. Aslan et al. [13] group such techniques in families:

- Encryption: malicious payload is encrypted to hinder static analysis
- Oligomorphic: asymmetrically encrypted
- Polymorphic: generalization of oligomorphic including several layers of encryption

- Packing: code is compressed and/or encrypted, potentially multiple times using ad hoc programs (called "Packers") to obfuscate it
- Stealth: malware makes changes to the infected system to evade detection attempts along with other techniques to prevent correct analysis
- Metamorphic: malware changes its code so that each sample looks different from the others [42]

Dynamic malware analysis platforms can be classified depending on the setup used to run the malware to be analyzed. Or-Meir et al. [42] distinguish between bare metal setups, virtualized, containerized, and emulated environments. In the first case, the malware is run on a physical machine with no interposed software layers between the malware itself and the hardware apart from the OS . Virtualized setups are based on virtual machines, the malware runs into a virtualized operating system and its behavior is observed from the hypervisor. Containers are isolation and control abstractions implemented in commodity OS es to isolate computational workloads and control their resource accesses. Containerization may be used to analyze malware while still isolating it from the host system. In the last setup, analysis is carried out using software emulation of the hardware used by the OS for software-based fine-grained tracing. Or-Meir et al. [42] also introduce techniques used to track malware activity, which include function call analysis, execution flow, and, memory and network activity tracing. Each technique may map to one or more kinds of setups. They are usually implemented by tracking programs that may fall into the potentially overlapping categories of debuggers, dynamic binary instrumentation tools, and sandboxes. Software debuggers are commodity software used to analyze and potentially correct the errant behavior of software artifacts, while DBI consists of adding analysis code to the analyzed program instructions to monitor its behavior. Sandboxing, instead, is about running a program in an isolated environment, potentially including analysis facilities to extract program behavior also on a very low level of abstraction. Virtualization-based sandboxes are a very popular setup [34]. In such cases, the malware runs either at the user or kernel level, while the monitoring solution runs at the more privileged hypervisor level. The analysis task may benefit from the privileged view over the running system, as demonstrated by several studies [22, 34].

Virtualization-based sandboxing for malware analysis is detectable by the malware itself analyzing virtualized hardware behavior and configuration [26]. Most hypervisors can emulate a vast variety of devices, granting behavior compatibility with existing software but the configurations of such emulated devices leak information about emulation itself. Moreover, hypervisors share TLB and hardware caches with virtualized guests. Since such

facilities are limited in size and shape, excessive TLB and cache misses may lead TLB-intensive applications in the guest to detect virtualization. Besides, time discrepancies due to virtualization may lead to the detection of the VMM and the sandboxed environment. If the guest reads the value of time-keeping model-specific registers (like TSC on intel x86 or even performance counters) to measure the time taken by a given instruction (e.g. a memory accessing one, a privileged instruction, etc.) the overhead resulting from virtualization interfering with the execution of such instruction can be detected. To counter its detection, the monitoring system has to proactively apply modifications to the view that the malware has of the running system to conceal its presence but such techniques still do not ensure that the VMM is entirely transparent[24, 26].

Malware samples use specific evasive techniques to perform virtualization detection and evade sandboxing, including:

- CPU fingerprinting: the behavior of some instructions is analyzed by the malware to detect anomalies caused by the presence of an underlying hypervisor (e.g. CPUID on x86)
- Timing analysis: precise time-measurement facilities in commodity processors may leak the presence of a virtual machine manager if the number of clock ticks and/or nanoseconds elapsed in between two instructions is too high because of traps in the hypervisor
- Registry analysis: if the system registry exposes entries that leak information about the VMM and/or the debugger, malware can easily note it
- System inspection: system device configurations and settings may reveal the presence of emulation or virtualization. The kernel module, as well, may be inspected to check whether a monitoring component is among them

Alongside such methods, malware in the wild also uses specific anti-debugging and anti-instrumentation techniques. The most popular ones are:

- Memory fingerprinting: program memory is parsed and analyzed by malware to check the presence of flags that may indicate the presence of a debugger
- Process environment inspection: the value of certain registers may suggest ongoing debugging actions (e.g. Debug registers may contain addresses of malware routines)
- Traps and exceptions analysis: if malware installs a custom exception handler but the debugger handles that particular exception itself, the fact that the handler

routine is never invoked suggests the presence of the debugger. Moreover, if the malware triggers a debug trap and program execution continues, debugger presence is revealed since it handled the trap.

As suggested by the description, some of these techniques can be used both to reveal the presence of a hypervisor and of a debugger [24].
The authors of [52] state that run-time packers are software routines used by malware authors to compress, encrypt, or obfuscate malware code. Several categories of packers have been identified in recent years, and what they have in common is that, before running, packed malware has to be unpacked, i.e. malicious code has to be deobfuscated and written to a memory location. After the termination of the unpacking routine, a "tail jump" diverts the execution flow to the unpacked region, which may either contain the full unpacked code or parts of it [52]. Being able to synchronize with potential unpacking attempts may provide high visibility on unpacked code, potentially easing analysis and detection tasks. Synchronizing with user and kernel events (stopping and restarting kernel operations) and preventing stealth evasive malware acting in both kernel and user mode from evading detection, imply that detection systems must work at a higher privilege level than kernel [22] limiting as much as possible the interference with the virtualized system not to be detected.

3 | State of the art

This chapter describes unpacking detection approaches using emulation, debugging, and instrumentation techniques, hypervisor-based monitoring solutions for solving such problem, and their performance implications.

3.1. Code unpacking detection

OmniUnpack [37] is a Windows Kernel driver implementing the "Write XOR Execute" policy in user-mode Windows processes to prevent executable pages from being writable and vice versa. Memory permissions are manipulated to spot unpacking attempts and subsequently try to infer when the unpacking of malicious code is complete. Upon completion, a user-space component analyzes the memory of the potentially unpacked malware. It works by marking a written page as nonexecutable so that subsequent execution attempts result in a protection exception in the OS which is caught and managed by the driver. Since not every processor had fine-grained permissions control at the time of the publication, memory permissions manipulations rely on software emulation. It only works with packed malware in user space, it cannot monitor kernel space effectively. Even extending to kernel memory, a kernel rootkit would be able to bypass the mechanism, the latter running at the same privilege level. Working with page granularity, it cannot identify the precise address of the page permissions violation, so all the page is parsed and analyzed. The system calls dispatch table is hooked by the driver to trap on syscall invocation and track the execution of the monitored program. The malware detector component performs signature-based detection. The authors define a signature scheme suitable for working with binary data with the granularity of a page. Page faults are factored out of the monitoring testing process forcing the operating system to load all the memory pages of the program upon loading, which is not a realistic context.

PolyUnpack [49] is a tool to automate the unpacking and disassembly of packed malware. It is a command line utility for Windows exploiting hardware and software breakpoints through Windows debugging APIs to debug and instrument the potentially packed binary.

Upon each instruction fetch, the tool disassembles the potential code block starting at the program counter address and matches it against a database created using static analysis made up of all the identified code sequences in the binary. If the instruction sequence is not already part of the database, it is considered new unpacked code. Debugger presence is hidden manipulating the operating system task control structure to remove the traces of debugging activity. The tool lacks stealthiness and debugger-aware malware can evade it since, as stressed by the authors, it is not realistic to block all the attempts of the malware to detect debuggers. The tools do not do packer malware detection in real-time.

Renovo [30] is a hidden code extractor based on TEMU, an emulator for binary analysis. The emulator creates an analysis environment for the guest operating system, sets up a CR3 trap to capture context switches, and to understand when the monitored process is scheduled to be executed by the guest. It observes the monitored process maintaining two views over its memory: standard and shadow view. Memory writes are instrumented by the emulator to trace the writes that happen in the memory of the process and, upon writing, the shadow memory is examined and the bytes written are marked as dirty into a bitmap managed by the emulated environment. Upon each instruction fetch by the monitored program, the emulator checks the address of the program counter of the program to understand whether it is overlapped with one of the written locations or not. In case of overlapping, hidden code is detected and dumped to the disk. Process instrumentation is supported by a driver installed in the guest operating system. This makes the system unsuitable for kernel rootkit analysis and potentially vulnerable to emulation detection by the malware.

Stealth Debugger [32] is a hypervisor for debugging and tracing memory accesses of a process contrasting the anti-debug functions of the malware. The main focus of the work is to effectively identify the full unpacked code among many candidates emerging from the analysis and then find its entry point precisely. To do so, execution has to be traced at a very fine grain. Stealth Debugger retrofits QEMU to implement stealthy breakpoints, memory tracing and to emulate time readings by the guest. An in-guest component is used to bridge the semantic gap and trace guest execution. Such a component is protected by manipulating guest kernel structures to block the operating system and, consequently, the processes from accessing the memory location where the driver is installed (Direct Kernel Module Manipulation). Memory accesses are traced at page-level granularity and execution attempts of written memory are captured acting on the MMU of Intel x86 processors but no implementation detail is provided. Code detection is performed using a scoring system to rank code candidates. In the best case, the time taken by code identification

is 429 seconds, which makes this approach unsuitable for real-time malware unpacking detection on commodity systems.

HyperDbg [31] is a modern hypervisor-based debugger for the Windows operating system. It leverages Intel VT-x hardware support to support the implementation of flexible scriptable debugging flows independently of the OS Debugging APIs. This makes it undetectable by malware checking for kernel debugging APIs footprints on a debuggee. The debugger is based on a hypervisor loaded in the guest on the fly. It communicates with a host-based user mode controller application which provides an interface to the HyperDbg scripting engine. This engine is VMX-root mode compatible and enables users to define custom debugging scripts that can be executed directly in the hypervisor. Some of the novelties behind HyperDbg, which make it substantially faster than state-of-the-art kernel and user debuggers, are the use of Monitor Trap Flag for precise debugging step-in and of debug registers for step-over and the exploitation of Intel Transactional Memory to cope with the semantic gap to check the validity of guest addresses without the need to directly traverse the page tables. System calls are hooked disabling the syscall and sysret instructions overwriting guest IA32_EFER register. Upon system call invocation, an Undefined Opcode exception is triggered and a trap in the hypervisor informs the debugger of the execution of a system call. VMI functions reconstruct the parameters passed to the API. Besides that, HyperDbg exposes hidden EPT-based hidden hooks through the CLI which can either be VM-Exit based or in-guest (Detours). The first type of hook is implemented by manipulating EPT permissions to generate traps in the hypervisor upon the execution of the guest memory operation of interest. The second type of hooks are fast, inline hypervisor-protected kernel-level hooks implemented patching the first 19 bytes of a routine to redirect routine execution to a hypervisor-controlled memory area with no VM-Exit. Upon detours execution, normal control flow is restored. The hook is protected by removing write and read permissions from the page containing the hooked routine and emulating read and write attempts upon them trapping in the hypervisor. EPT hidden hooks are used to implement debug register emulation so that an unlimited number of such registers can transparently be used to track events of interest across a vast range of guest addresses at once. HyperDbg breakpoints are implemented using EPT hooks as well. Whenever a stealthy breakpoint is needed, the page where the breakpoint should be placed is duplicated, and execution permissions are removed. The duplicate page is neither writable nor readable but executable. The breakpoint opcode of x86 processors is written in this latter at the location specified by the user. Upon hit, a trap to the hypervisor happens since the guest is configured to trap into the VMM if a breakpoint exception is raised. Reading and writing the duplicated modified executable page results

in a trap to the hypervisor which emulates the operation.To prevent the debugger from being detected using time-deltas methods as described in 2.1.6, guest access to performance counters and time-related MSRs is blocked by configuring the VMCS to trap into the hypervisor upon TSC and Performance counters readings. The offending instruction is emulated and values returned to the guest are extracted from a statistical distribution which is obtained upon training the debugger on timing data coming from the guest OS, collected right before the activation of the hypervisor-debugger. Complex hiding actions are implemented in the "Hide mode", which can be activated at any time by the user. The authors showcase the tool for rediscovering an old Windows 0-day bug and suggest to use HyperDbg for forensics as well.

3.2. Hypervisor-based monitoring

This section summarizes the state-of-the-art reporting criteria to create a taxonomy of monitoring solutions based on hardware and hypervisor facilities that leverage Virtual Machine Introspection to fill the semantic gap described in 2.1.4. These criteria will be used to characterize the solutions which have been tested in the present work.

Hebbal et al. [28] trace a neat separation between in-VM and out-of-VM methods. The former leverage in-guest components coordinated with the hypervisor to gain insights into the guest OS state. The latter either exploits OS source code and debugging data to solve the problem of the semantic gap or relies on hardware events. Out-of-VM techniques relying on both strategies are called "Hybrid". Both in and out-of-VM approaches can coexist and the suite Napoca + HVMI by BitDefender is an example of such an approach [2]. The present thesis work focuses on out-of-vm approaches, as malware may spot monitoring presence when running in the same context as the monitor as stated in Section 2.1.6.

A further criterion to classify hypervisor-based monitoring solutions is in-band vs out-of-band solutions. The first type of solution reads the memory of the virtualized guest without stopping its execution. This practice is not invasive to the execution of virtualized software, but it may introduce inconsistencies in reads from the memory of the virtual machine since the state of the guest is dynamically changing during the monitoring. This makes it not suitable for fine-grained synchronous execution control over the guest. The second type of solution is more invasive than the first concerning the guest execution since exits to the hypervisor interfere with the execution of the virtual machine, but it is suitable for synchronizing the monitoring flow with the guest [50]. Even though a large variety of events can be tracked at the hypervisor level, the present work focuses on

tracing memory operations, with emphasis on code unpacking attempts detection. In the following, solutions specifically solving those issues are presented along with works that may help justify the need for efficient unpacking detection solutions for malware analysis and detection and works discussing the performance of existing solutions.

Ether [22] is an extension to the Xen hypervisor for malware analysis and sandboxing using VT-x by Intel. The hooking system of the tool can both handle system call interception and fine-grained instruction fetches and memory writes tracing. Memory writes are captured leveraging hypervisor protection at the Shadow Page Tables, while instruction fetches are traced using Monitor Trap Flag. Ether retrofits software-based virtualization systems filtering out the page faults trapping in the hypervisor resulting from legitimate guest operations passing them to the guest, while it directly handles page faults resulting from writes in particular locations. Such exceptions are used to have fine-grained control over the guest memory writing activities. Syscalls are traced by setting intel Model Specific Register SYSENTER_EIP_MSR to the address of a page which is guaranteed not to be in memory. Upon the execution of SYSENTER, the guest jumps at such location and causes a page fault at the specific address established before which is intercepted by Ether to track syscall activity. Similarly, legacy software system invocations leveraging interrupts are handled by injecting page faults upon jumps on the IDT entry responsible for system call handling. Ether conceals MSR modifications trapping in Xen at each guest attempt to manipulate such registers. The authors of Ether propose Ether-Unpack, a framework for malware unpacking detection and analysis. The authors of Ether state that this work is not intended to be used for real-time analysis since the guest suffers from excessive trapping-induced slowdown but they do not provide data on the performance overhead incurred by the guest during the execution of the tool.

Patagonix [36] is a library running on top of Xen that detects code subversion and hidden code execution using the NX bit of the MMU of modern intel processors. NX stands for Non-Executable and it is a hardware-based mechanism used to prevent the execution of certain memory regions. Page Table entries with the NX bit set to 1 are not executable and instruction fetches from such pages raise exceptions, potentially trapping in the hypervisor if the guest is configured to do so. In Patagonix, execution attempts over the protected page are monitored and authorized only upon verification. Verification consists of matching the code to be executed against an internal database of trusted code. In case of no match, the user is alerted. Upon guest initialization, Patagonix activates the NX bit on each page allocated to the guest. Whenever pages are executed for the first time, their content is checked according to the logic above described. If the content of

the page is legitimate, the NX bit is unset to impede subsequent traps in the hypervisor, limiting performance overhead. At the same time, write permissions to the executed page are removed so that subsequent writes to the page will be captured. Upon a new write to an already checked page, NX is reset, and monitoring restarts. At the time of publication, Patagonix also monitored for packing and could unpack packed binaries, but only if packed with the popular UPX packer, which is the main limitation of the tool.

NICKLE performs real-time kernel code authentication to prevent kernel rootkits from executing[48]. Execution attempts are captured by trapping in the hypervisor upon instruction fetch using Memory Shadowing at the hypervisor level. NICKLE interfaces with the hypervisor to create two views of the physical memory of the guest: standard and shadow memory. Shadow memory only contains authenticated code, copied there upon guest kernel initialization, and, in case of instruction fetches, execution is transparently redirected by the hypervisor to the shadow memory. Memory accesses other than execution are handled using standard memory view. If a kernel rootkit tries to write its code to the standard memory and then execute it, the hypervisor compares standard and shadow memory upon execution of the just written routine and it can detect the inconsistency. NICKLE has been developed as an extension to modern hypervisors like QEMU+KVM, VirtualBox, and VMWare. NICKLE does not monitor user space and incurs substantial overhead for memory duplication. No technical implementation details are provided by the authors.

CXPInspector [55] brings the idea of memory partitioning to the hardware-based virtualization realm. It alternates between multiple guest memory views to capture execution attempts in monitoring the virtual machine using Intel EPT. Starting from the standard EPT memory view of a guest, CXPInspector establishes a duplicated EPT view and then synchronizes both the original and duplicated versions. Subsequently, the implemented CXP mechanism dynamically partitions guest memory into executable and non-executable memory segments. When the instruction pointer references non-executable memory, a VM-Exit is triggered and the hypervisor takes control from the guest OS for inspection purposes. This idea is at the base of stealthy EPT hooks implemented to intercept guest APIs invocation and extract calling arguments and return values. Virtual machine introspection is used to reconstruct the state of the guest, solving the semantic gap problem. The authors of [55] propose its use for malware (kernel rootkit) reverse engineering.

Drakvuf [34] is a malware analysis library extending LibVMI to implement hypervisor-based sandboxing on top of Xen through Virtual Machine Introspection. On loading, the

library defines multiple EPT views for the monitored virtual machine leveraging hardware support as described in 2.1.2 and extensive software support provided by Xen via the **altp2m** feature. Altp2m is an enhancement enabling Xen to effectively exploit VT-x multiple page table per guest handling capabilities. The installed multiple EPT views and the advanced hardware features like execute-only EPT pages enable the extensions of the library to implement fine-grained control over guest actions. The great advantage of altp2m is represented by the possibility of entirely avoiding implementing ISA emulation inside the hypervisor and, since emulators may introduce behavior discrepancies because of the impossibility of closely reproducing the behavior of closed hardware architectures, altp2m is a game-changer [33]. The architecture of Drakvuf is modular and easily extensible with plugins implementing custom tailor-made monitoring and introspection actions. Existing plugins exploit Drakvuf APIs for IDT manipulation and monitoring, for tracing reads/writes/executes from a specific memory location, to define stealthy breakpoints to debug and monitor both kernel and user mode, and for many other use cases. Drakvuf works by hooking kernel functions to follow the execution of the guest, tracking process creation, termination, memory allocations, and kernel memory layout manipulation closely. Hooks are dynamically installed in the guest exploiting page table views switching at the hypervisor level on top of altp2m. In the case of stealthy breakpoints, the guest VM is set up to trap in the hypervisor upon breakpoint exception. Subsequently, software breakpoint opcode is written to the memory location of interest to the tracing flow, and, upon hit, a VM-Exit happens. The breakpoint is stealthy and invisible to the guest since it is only written into the executable view mapping of the specified page. The readable/writable view mapping is left untouched. By default, the executable view is the used one. If the guest tries to read or overwrite the monitored location, upon trapping in the hypervisor, a callback restores the readable/writable memory view and the guest cannot see the modification or overwrite it. For coarse-grained page access monitoring, stealthy EPT hooks are used. Differently from breakpoint hooks, they revolve around permissions modifications that are relative to an entire memory page. For example, to track instruction fetches in the page execution permissions are removed from it installing a suitable EPT view. Whenever a permission violation in a random location of the page happens, the guest traps in the hypervisor. They are used to track page-level activity immaterially of the in-page offset where the violation occurred. System calls execution is tracked using stealthy breakpoints on system call handlers. It supports both Windows and Linux introspection and relies on kernel debug information to solve the semantic gap problem at runtime and gain insights on guest kernel and user events, kernel APIs, structures and routines offset with respect to its base. The Codemon plugin enables reverse-engineering oriented memory events analysis with EPT-based stealthy hooks on Windows. The kernel

memory management API MmAccessFault is hooked to synchronize with process memory accesses and then hooks are installed to track writes to accessed pages and potential subsequent executions of the written code. It may help detect unpacking and may dump unpacked code to host operating system-controlled memory. The lack of concrete facilities to effectively handle page swapping is the most important limitation of the tool, as this may cause instability in the guest OS.

Leon et al. [35] propose an unnamed tool for hypervisor-based guest userland processes sandboxing for malware detection. Their system is implemented as a UEFI application loading a hypervisor upon Windows initialization. The hypervisor hooks the system calls performed by the guest using in-guest detours. A detour is a particular type of hook whose handling code is in the guest, protected, and concealed by the hypervisor. The hypervisor is closely integrated into the guest kernel, which allows it to even call kernel APIs from the VMX root context. EPT and IOMMU are used by the hypervisor to hide the monitoring infrastructure. SLAT blocks the guest from accessing hypervisor memory removing read permissions and EPT permissions violations result in instruction emulation by the hypervisor for fine-grained read, write, and execute control over the guest. System calls hooking and fine-grained control are configured with a declarative textual file defined by the user for each process to describe the system calls to be hooked, their input parameters, and the rules for data collection. Upon system call invocation, the hypervisor records all the data relative to the call gathers them into an internal buffer, and progressively stores it in a disk location. Upon collection, data are processed by a Process Behavior Analyzer which converts them in JSON format to enable human analysis. The hypervisor does not implement malware detection tools and only monitors system call invocations, without implementing any fine-grained memory monitoring operation.

VMShield [40] is a toolchain based on Drakvuf for Machine Learning-based malware analysis. A Xen virtual machine is used to extract features describing the behavior of malware samples using introspection-powered system calls and memory hooks installed using LibVMI and Drakvuf. Upon data extraction, the sequences of system call invocations, memory writes and instruction fetches are given as input to a machine learning algorithm that uses the Bag-of-n-grams approach to generate malware features. The most relevant features are identified and retained over the feature set using an algorithm called Particle Swarm Optimization and the Machine Learning algorithm Random Forest classifier classifies the extracted behavior to detect malicious processes and tell them apart from benign ones. This work mainly focuses on the data collection and analysis part, making it unsuitable for efficient and real-time malware detection. Nonetheless, it presents a poten-

tial use case of virtual machine introspection in the field of malware analysis and detection.

HVMI is an introspection-based operating system hardening and malware protection framework developed by BitDefender. It implements hypervisor-based monitoring on top of several hypervisors such as Xen, KVM, and the BitDefender Napoca hypervisor. The Introcore library at the base exposes APIs to implement fine-grained memory tracing like Drakvuf through EPT permissions manipulation but without creating extra EPT views for the monitored guest [2]. This can be done by altering page permissions at the SLAT level, i.e. inside the hypervisor, which makes the operation invisible to the interested guest. Other memory modifications are not visible to the guest thanks to memory cloaking. Read and write permissions are removed for the specific page table entries containing the mappings for modified guest memory portions and read/write attempts eventually trap in the hypervisor. Kernel functions are hooked to follow closely guest execution and traps are checked against a rule-based internal integrity protection subsystem. Such functions are located in kernel memory linearly searching executable sections and performing signature matching against the signatures the library programmatically extracts from the kernel binary at configuration time. Kernel hooking is implemented using detours. When triggered, a detour deviates the execution of kernel-level code to a hypervisor-controlled memory region in the guest kernel. Such a region is made up of all the slack space found in kernel pages (i.e. the space not used by kernel code in executable pages). HVMI writes detours handling code in such unused memory regions, chains them together, and protects them against analysis and tampering by the guest kernel using memory hooks on slack locations. This enables efficient, trap-less kernel monitoring to solve the semantic gap and catch interesting events (process creation, destruction, heap allocations, and so on). The framework is extensible with in-guest agents to optimize introspection and enable complex monitoring flows. Agents can be of several types and, from a performance standpoint, the most important is the Virtualization Exception-based agent for efficient exit-less handling inside the guest of EPT violations caused by the activity of the operating system pager over the hooked page table entries. HVMI implements hooking directly acting on the memory pages containing page table data structures and this enables support for seamless page swapping handling put in place by the guest operating system. HVMI provides multiple facilities to hook system calls, such as CR3 hooks to detect the transition between kernel CR3 and user mode processes CR3, EPT-based hooks, or in-guest detours installed directly on the handlers in kernel address space. User space is realized by tracking kernel functions, and library functions inside Windows user-space runtime libraries, and hooking kernel and user mode data structures. HVMI supports code unpacking detection, but it is limited to userspace memory. HVMI can either be deployed in the host userspace, inter-

facing with the underlying type 1 hypervisor using compatibility layers and kernel-based communication facilities (a driver), or can be directly integrated into the hypervisor code statically linking it at build time and deploying it in a privileged host disk location.

3.3. Performance-oriented works

Virtual machine introspection solutions performance has rarely been investigated in detail. Existing approaches have performance problems and may be invasive to the guest execution [50]. That is why there many attempts have been made to bring more efficiency in the VMI realm. In the following, works on performance assessment and optimization of Virtual Machine Introspection solutions are presented. What emerges from the analysis is that solutions for supporting code unpacking and hidden code extraction have never been compared on a performance level, but just from a functional standpoint.

Bauman et al. [14] survey and discuss hypervisor-based monitoring solutions. A taxonomy is defined to point out the characteristics of the various solutions, passing through the way the semantic gap problem is solved and whether or not the monitoring leverages in-guest components. In this work, the limitations of existing solutions are pointed out and, in particular, the lack of standardization for benchmarks and tests against which the performance level of existing solutions can be assessed is discussed. Authors also add that since existing solutions are oftentimes based on different hardware and software core infrastructures, defining evaluation metrics that are expressive enough, but still generic to abstract from the backend variability, is crucial. No quantitative evaluation of existing VMI-based guest hooking and monitoring solutions is provided.

Exploring VM Introspection: Techniques and Trade-offs [50] aims at systematizing the knowledge in the area of VMI comparing existing approaches up to 2015 from a functional and performance standpoint to guide design space exploration in VMI architectures. A taxonomy of VMI solutions is provided, distinguishing solutions that work on memory dumps from live forensics-oriented ones and classifying them based on the type of guest memory access implementation at the base. Particular focus is posed on the difference between guest-halting solutions, which suspend the guest upon certain events to read its memory, and those who do not. The major claim is that despite guest-halting solutions having a quite significant overhead over the execution of virtualized workflows (due to frequent suspension), they are not able to mitigate all the inconsistencies that VMI solutions suffer from stemming from the semantic gap. At last, the authors propose several metrics against which the examined passive VMI solutions should be compared, establish

benchmarks for VMI solutions, and then present a quantitative comparison of such solutions. The main limitation of the work is the focus on VMI for guest memory reading operations (passive VMI [21]), which does not entail monitoring-oriented flows based on stealthy hooks, emulation, and guest memory views manipulation (active VMI [21]).

RapidVMI is an extension to the LibVMI framework aiming at solving some of the performance issues of VMI-based monitoring systems [21]. Its contribution is threefold: introduce process-bound events injection, implement core-selective code injection, and reduce the slowdown resulting from VM-Exits management via LibVMI on the Xen hypervisor. The first objective is pursued by employing a separate EPT view for the sole monitored process. Upon CR3 write, a VM-exit is executed and, if the CR3 value is associated with the monitored process, the hypervisor changes the EPT view to the one used for it and sandboxes it. Every memory modification to the memory view of the process is only visible to it, but this potentially requires the duplication of the full memory of the process, including shared memory pages. A similar strategy based on altp2m and memory duplication is used to restrict code injection to one of many cores running code in a shared memory location. The third objective is realized by implementing CR3 writes-related and EPT violation traps handling directly in the hypervisor. The short-circuiting of such events allows the reduction of the time for which the guest is stopped of up to 98 times and it shows that managing such events at the hypervisor level, without passing them to the host operating system and then to the user space is substantially more convenient to potentially enable real-time use cases of virtual machine introspection. The main limitation of the work is the fact that OS-level paging in the guest is disabled and the authors make the strong assumption that all the pages of the given process are already mapped in memory at process beginning. Moreover, the proposed optimizations apply to the case of monitoring restricted to a single process, not to the overall system.

Abdelraoof et al. [12] propose an optimization for LibVMI on top of Xen. They precisely describe the synchronization happening between virtualized guests trapping in the hypervisor, hypervisor-level trap handler, and the user-mode introspection library running in the host. During the handling of the callbacks registered in the library to manage guest events (e.g. system calls and CR3 writes) the hypervisor is called several times either to switch the guest page tables or to map for read/write guest memory. Since such transitions introduce a substantial overhead, the main contribution of the paper is about factoring out of the monitoring flow hypercalls invocation, placing in-hypervisor traps management components to implement memory tracing and breakpoint handling logic at ring -1. This optimization, however, has the drawback of limiting the flexibility of VMI

libraries, since substantial changes involve recompilation and redeployment of the hypervisor. Then, to limit the time window in which the guest operating system is stopped, they introduce asynchronous inter-domain event message passing for Dom0 to Xen and vice versa communication. This implies that, if Dom0 does not consume events at a rate faster than production by Xen, full observability over the events generated by the guest is lost. Moreover, a synchronous answer to such events is no longer possible. This lowers VMI overhead but also hampers the synchronous interaction needed for monitoring the guest.

4 | Problem statement

This chapter introduces the research questions we posed ourselves in the present thesis, the goals of our research, and the challenges we had to face.

4.1. Research questions

In the state of the art, there is no work strictly focused on the sole task of code unpacking detection at the hypervisor level. Existing solutions and various libraries implement facilities for deep state reconstruction on virtualized operating systems, fine-grained memory activity tracking, and execution control. They revolve around the concept of memory hooks, which are points in the memory of the guest used to capture certain types of memory accesses.

The placement of the hooks inside strategic observation points is at the base of the synchronization of monitoring solutions to the execution of the guest. Identifying such points entails reconstructing the state and there exist several strategies to do so, some leveraging extensive knowledge of monitored OS internals, others limiting their assumptions on the execution of the guest to a bare minimum.

Based on the different hooking and state reconstruction strategies, we asked ourselves what is the impact from a performance standpoint of tracking memory accesses done by a virtualized operating system at the hypervisor level. We posed the questions of how hypervisor-based code unpacking detection impacts the performance of memory accesses in a monitored virtual machine, and how hypervisor-basesd monitoring impacts the execution of complex real-world applications. In this context, we investigated how alternatives for hypervisor-based monitoring compare to each other, trying to find out how an approach that assumes a deep knowledge about the monitored operating system compares to a simpler one, only leveraging partial knowledge of OS internals.

As a secondary research question, we enquired whether a hypervisor-based debugger can be retrofitted as a memory monitor aimed at detecting potential code unpacking. We wanted to know whether the approach used to track changes in the memory of a virtual

machine employed by existing solutions is reproducible from a functional standpoint on such a debugger and, if so, assess the performance overhead resulting from its use.

4.2. Goals and Challenges

The main goal of this work is to investigate the performance overhead on processes running in a virtual machine caused by hypervisor-based monitoring solutions, comparing different approaches to memory access tracing and guest state reconstruction.

Our starting point is the comparison between two alternatives for memory tracing based on memory permissions alteration and permissions set switching leveraging Intel EPT, described in Chapter 3. The state of the art lacks comparisons from a performance standpoint of the different approaches to memory tracing at the hypervisor level via the installation of memory hooks. We selected two approaches, one based on multiple memory views and page table pointer switching, and the other based on permissions removal and emulation. The main challenge to their comparison is the design of a micro-benchmark that stresses a specific memory access pattern in an in-guest process. Providing an estimate of the overhead of memory hooks management and swapping upon state changes is important to establish a metric to compare different hooking approaches and possibly guide design-space exploration of monitoring solutions. We chose guest program execution time as such a metric, focusing on the slowdown of guest operations. This choice implies that platform variability has to be taken into account to grant consistency in time measurements across different hypervisors and to ensure that measurements are comparable.

One of our goals is to measure the overall slowdown incurred by real-world interactive applications, to understand whether hypervisor-based monitoring for memory tracing is a viable option for real-time system protection and hardening. We selected two approaches to test and compare, one implemented on the Drakvuf VMI library and the other one on the HVMI OS hardening framework. Turning to real-world programs of common use, the main problem is the simulation of user interaction, which can be solved through the automation of user interface operations. One challenge deriving from automation is the need for robust control over variable latency in the interaction with graphical components due to the aleatory slowdown of guest functioning caused by the hypervisor. Upon solving this problem, a realistic macro-benchmark can be designed to capture user behavior.

Last, the secondary goal of this work is to evaluate the memory monitoring capabilities of the hypervisor-based debugger HyperDbg. The novelty behind this debugger is that it integrates a scriptable interface to the hypervisor itself, with efficient exits to the VMM

handling. We want to know how trap handling at the hypervisor level with HyperDbg compares from a performance standpoint to other solutions, focusing on the overhead incurred by the memory operations of a monitored virtual machine, an absent datum in the literature. Before evaluating its performance, we have to design a script to reproduce from a functional standpoint the monitoring flows implemented in Drakvuf and HVMI. Despite having a flexible script engine, HyperDbg is not created with monitoring in mind, and finding a scheme for implementing memory tracing for code unpacking detection is made challenging by the lack of dynamic hook-switching capabilities.

5 | Approach

This chapter describes at a high level of abstraction the problem of unpacking, repurposing a formalization of the task of code unpacking taken from [52]. Subsequently, it presents the methodology used to investigate hypervisor-based monitoring performance.

5.1. Code unpacking overview

Popular run-time packers are associated with potentially very involved unpacking routines used to extract the packed code before its execution. The code unpacking process articulates in various phases, whose relative order is not fixed, and, usually, the unobfuscated code of an executable is not entirely visible in memory simultaneously. For the sake of abstraction and simplification, code unpacking can be modeled starting from the formalization of the process for unpacking one region of a packed executable provided by Ugarte Pedro et al. [52]. They used the proposed model to implement a tool for packed binaries inspection, tracing memory operations at the byte granularity. In our case, we decided to work on a coarser grain, observing memory evolution at the page granularity as this naturally allows us to exploit modern hardware facilities like page-level permissions manipulations as described in section 2.1.1.

Our model does not take into account some state transitions that may be of interest for the packer malware domain, such as the re-packing of malware code. This choice is motivated by the fact that we do not delve deep into the packer malware phenomenon. Instead, we try to understand the overhead associated with tracking state changes in **executable** guest memory, focusing on *write accesses to executable memory followed by instruction fetches* from the just written memory location. Our simplified model captures the transition of one page P from a "clean" state C to a "dirty" state D and from D to U standing for "unpacked".

The starting state is C. It represents the context in which, after the allocation and the initialization (e.g. to copy the code of a process), no write accesses to the page P happened. Upon the first write, the page transitions to D. D stands for "dirty" as now the code

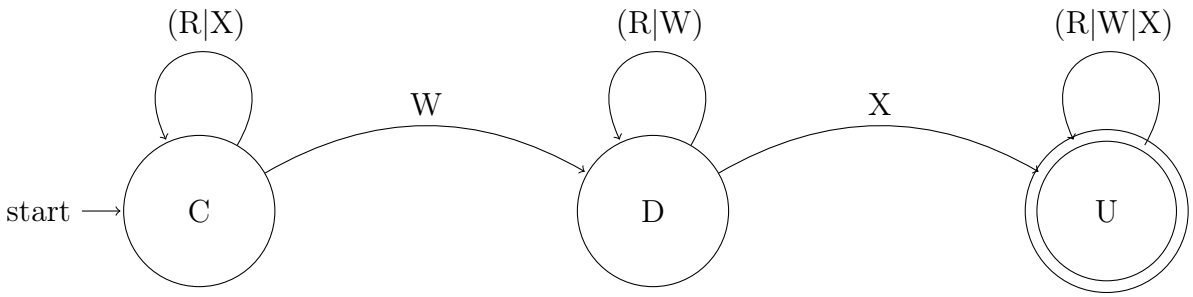

Figure 5.1: Finite state automaton describing state changes of monitored page P

previously written on the page may have been modified by the writes. If an instruction fetch happens on this page, we assume code unpacking potentially took place. On this event, the page state evolves in U, which stands for "unpacked". If blocking the execution of unpacked code is the objective, this state becomes absorbing in that subsequent accesses to the page are prevented.

The Finite State Automaton in Figure 5.1 summarizes what just said and pictures the evolution of the state of a memory page during code unpacking. Possible accesses to a page are read, write, and execute.

Symbol	Meaning
R	read access
W	write access
X	execute access
C	clean
D	dirty
U	unpacked

Table 5.1: Legend to Figure 5.1

5.2. Detecting code unpacking at the hypervisor

Based on the formalization of code unpacking proposed in section 5.1, this section introduces our approach to set up monitoring actions at the hypervisor level used to enable synchronization with the memory operations of the guest aiming to detect potential code unpacking attempts.

We restrict the monitoring to one single process at a time. This choice is motivated by the fact that we want to characterize the performance penalty of the single process. In a monitored virtualized system, activities inside the guest operating system other than the

ones directly linked to the execution of the monitored process may generate traps and false positives in the monitoring infrastructure getting in the way of the precise performance assessment task. To avoid this, isolation is enforced by leveraging ad hoc primitives at the hypervisor and introspection library level, presented in chapter 6.

To closely monitor a single process, we extended the model for page access tracking in Figure 5.1 to include monitoring actions. The extended model is presented in Automata 5.2. Monitoring starts with the tested tools installing a memory hook on page P allocated to a monitored process with execution permission enabled. Depending on the used monitoring tool, executable page discovery may happen at process startup when the virtual address space of the process is set up, or during its runtime, as soon as an executable page is accessed.

In the first case, upon the discovery of **executable pages**, the tool hooks against write operations these latter, forcing permissions violations to trap in the hypervisor. This action forces a transition bringing the page into the "clean" state. In the second, an execute hook is installed on each page accessed by the process, and the hooks that fire inform the hypervisor that the page where the violation occurred may contain code. The hypervisor manages this event **swapping** the execute hook with a write one, bringing the page into "monitored" state.

A trap on the "clean" page P results in the installed hook being **swapped** with an execute one and page P undergoes a transition from "clean" to "dirty". The write hook is removed after the swap because, in our model, a "dirty" page is still "dirty" after a write to it. If the guest tries to fetch instruction from a "dirty" page hooked against execution, the page state changes to "unpacked". If blocking the execution of unpacked code is the objective, this state becomes absorbing and subsequent accesses to the page are prevented.

The automaton in Figure 5.2 depicts page state transitions and relative monitoring action in the format "event/action" on the edges.

5.3. Benchmarking existing solutions

Since Drakvuf makes large use of the Xen altp2m feature, which is not used by HVMI, in our approach to investigating the performance of the two solutions we include the characterization of the slowdown introduced by EPTP switching at the hypervisor level. This is a preliminary task to the presented analysis, shedding some light on how costly such a practice is in terms of time. That being said, our approach for assessing the performance overhead associated with hypervisor-based monitoring entails two tasks. First, we want

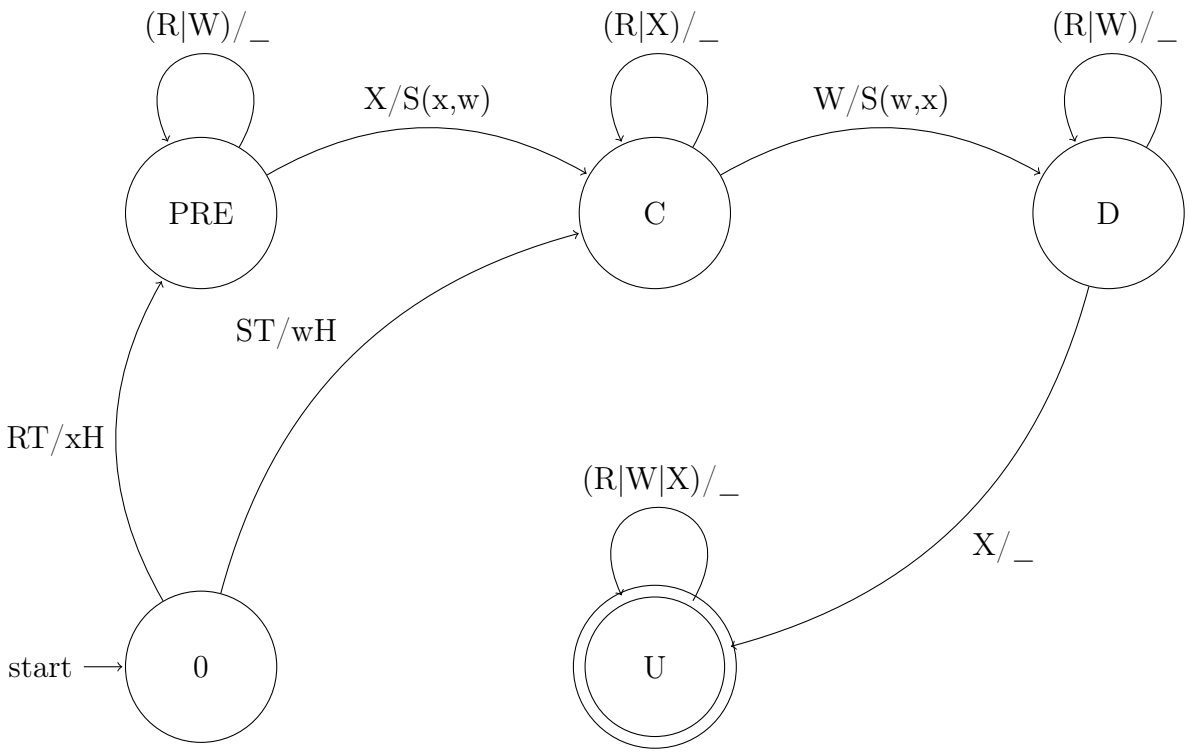

Figure 5.2: Extended finite state automaton describing the monitoring flow for page P

to characterize the slowdown introduced on memory operations by the **trap switching** logic in the hypervisor used to track the activities on a given memory page. To accomplish this goal, we wrote a **micro-benchmark** program intended to run inside a monitored virtual machine which executes a routine designed to stress the proposed hypervisor-based memory tracing facilities. Such routine writes and executes instructions from the same memory location in a loop. We call it a micro-benchmark since we investigate the performance penalty on memory operations carried out by single instructions, which we think are the simplest abstraction to reason about in this work.

Stressing such a precise memory access pattern, we are able to describe quantitatively the impact of hypervisor-based memory hooking on the execution of memory accesses in the worst-case scenario in which every memory access causes a stop the execution of a virtual machine.

Secondly, we aim to investigate the **overhead on real-world applications** of hypervisor-based monitoring. Despite such applications do not exhibit code unpacking behavior, dynamically managing process lifecycle tracking and memory hooking facilities introduces a certain overhead on the process. We developed a **macro-benchmark** program to assess the slowdown in process execution resulting from the activation of hypervisor-based monitoring on the process of interest. Such macro-benchmark exploits Windows Graphical

Symbol	Meaning
R	read access
W	write access
X	execute access
C	clean
D	dirty
U	unpacked
kH	hook P for permission "k"
S(x,y)	swap trap "x" with "y"
_	null action ("do nothing")
ST	P known to be executable at process startup
RT	permissions of P not known at startup

Table 5.2: Legend to Figure 5.2

User Interface automation facilities to mimic real user interaction with commonly used Windows applications. We called it macro-benchmark since we benchmark the execution of real-world applications, not focusing on simple memory operations.

Testing benign real-world applications allows us to quantitatively describe the overhead on their execution resulting from tracking state changes inside the virtual machine at the hypervisor, putting us into the average-case scenario of monitoring everyday systems. This help us to investigate the feasibility of hypervisor-based real-time unpacking detection.

5.4. Evaluating the cost of altp2m on Xen

Leveraging SLAT permissions modifications to implement memory hooking has long been done in history, as suggested in chapter 3, for security reasons. The use of multiple EPT views with different permissions set, instead, has been included as a feature in modern hypervisors quite recently (2016) with the Xen altp2m case [33]. Altp2m solves the shortcoming of instruction emulation in many contexts, allowing introspection solutions to be safely and easily implemented [33]. Such a feature, however, is still undergoing consistent testing activity and the maintainers of the Xen project still classify it as a "Tech Preview", suggesting that it is not production-ready yet [56]. However, as stated by the authors of [34], it has the potential to improve the security of modern virtualization-based systems, without the need to resort to unsafe alternatives that emulate the behavior of closed hardware architectures and potentially introduce incompatibility and behavior discrepancies. Moved by the need to explore how it influences the execution of guest operating systems, we decided to test it and assess the overhead resulting from the EPT pointers switch at the hypervisor level. This decision is also motivated by the fact that we

are currently exploiting this feature in the tested approach to hypervisor-based memory monitoring using the Drakvuf introspection library. To assess the overhead associated with altp2m monitoring we hooked one of the routines of the Xen hypervisor responsible for implementing page-table switching. On a very high level of abstraction, it switches the pointer to the currently used EPT view with the one to the new EPT view in each vCPU of the guest virtual machine. All the vCPUs are stopped before the operation. We collect time measurements before the invocation of the routine implementing the switch and right after its return. Time is measured leveraging the highest-resolution clock available to the hypervisor. The differences between time samples collected at the beginning and the end of the switching routine are averaged and the mean is compared to the average time taken by one iteration of the micro-benchmark. This is done to provide some potential insights on the order of magnitude of time taken by the Xen hypervisor to switch page tables view and just understand how it compares to time measurements relative to events happening inside the guest virtual machine. Even though the same time source is used to collect measurements in both cases, the implementation of time collection is potentially different since in the case of the micro-benchmark the C++ runtime is responsible for it. In the case of Xen, time is collected using wrappers on assembly routines also used for internal timekeeping in Xen. However, we hope our analysis facilitates the understanding of the overhead associated with altp2m and its potential impact on the performance of the guest operating system when used for active monitoring by Drakvuf.

5.4.1. Micro-Benchmarking

The synthetic micro-benchmark program we designed writes inside its process code memory area and then fetches instructions from the just-written location. Precisely, one instruction is fetched and the very same instruction is overwritten. Such instruction has been chosen to be a RET, which means that the shellcode returns to the caller right after being called. The execute and write sequence is performed in a loop for a number of iterations fixed through empirical observation to be high enough to have a stable estimate of the mean per-iteration time, while also keeping testing time reasonably low. We will refer to this loop using the symbol $\mathbb{W} + \mathbb{X}$ from this point on.

Subsequently, we extract the mean per-iteration time taken by such a process running in the virtualized operating system with and without enabling the described memory operations tracing at the hypervisor level adopted in our approach and presented in section 5.2. Finally, the two measurements are compared and the actual slowdown incurred by the tracked memory operations is computed. Time collection happens in the micro-benchmark program running in the context of the guest operating system, because taking

time measurements in the hypervisor may hamper the validity of the comparison among the various hypervisors due to platform variability. Two time measurements are taken in the microbenchmark: one before the execution of the loop of interest and one right after. The time taken by the loop to run is the difference between the last and the first measurement. Before starting the data collection process in the presence of hypervisor-based monitoring, the monitor is activated for the virtual machine of interest. After that, the micro-benchmark process is started and the monitoring infrastructure becomes aware of the guest virtual address of the memory area containing the code which is going to be overwritten during the execution of the micro-benchmark. Such information is captured synchronizing the monitoring flow with the process creation and execution in guest.

The core challenge faced while developing the micro-benchmark is the implementation of **synchronization** among the monitor at the hypervisor level and guest program execution. The semantic gap described in the section 2.1.4 of chapter 2 makes guest program execution monitoring problematic, since by default the bare hypervisor has no way to figure out what is the internal state of the computation in the guest and what is the memory layout of the monitored process. To address the semantic gap problem, introspection solutions were used to figure out where the interesting memory location is and to track its manipulations. Simply put, this boils down to installing hooks in suitable program locations and doing some filtering upon exits in the hypervisor in the introspection library itself. The number of installed hooks has been kept to an absolute minimum to limit the overhead on guest execution and simplify the monitoring process. Upon the triggering of each hook, the memory monitor synchronizes with guest program execution and extracts state information. Getting to know the address of the manipulated memory location, given the randomization of process address space in modern operating systems, is the biggest challenge to solve to set up tests for precise performance assessment. Nonetheless, the memory location has been extracted reconstructing the parameters passed to hooked Windows kernel memory allocations functions at invocation, leveraging reconstructed guest state data, knowledge about Windows API internals, and filtering out the data at the level of the introspection library. The approaches that one can use to perform these operations are strictly dependent on the introspection library and its capabilities. For this reason, some differences exist in the way the monitoring flow has been implemented by Drakvuf and HVMI.

On a very high level, Drakvuf hooks a system call, not just a generic kernel function. This system call is invoked in the written micro-benchmark to **change the memory permissions** of the memory area containing the interesting code to be analyzed (the loop above mentioned). It precedes the $\mathbb{W} + \mathbb{X}$ loop execution and one of the parameters

of the invocation is the virtual address of the memory page containing the code. After trapping, the hook is removed to prevent further traps from happening. The address of the page to monitor is extracted from the guest reconstructing the system call parameters at the library level after the trap in the hypervisor. Finally, the **write hook** is installed on that page after the extraction, and the monitoring starts following what is described in section 5.2. This approach has been motivated by the need to keep the monitoring flow as simple and lightweight as possible, reducing noise on the measurements and limiting performance degradation on the guest.

HVMI, on the contrary, can track process creation and follow closely the allocation of the virtual address space of the process as the authors designed the tool to be strictly integrated with Windows process creation functions. The library has ad hoc APIs for activating per-process protection against several types of attacks and hijacking attempts. In our approach, we set unpacking protection for the microbenchmark process, before it is started specifying the name of the executable. As soon as the process is created, HVMI detects that it has been set to be protected and handles this setting by installing all the necessary hooks. In particular, hooks are installed over the page table locations which are associated with the part of the virtual address space of the process that is marked as executable inside the header of the Windows executable. When such entries are populated by the operating system and the pages are physically allocated and initialized, HVMI modifies the permissions of the pages **acting at the hypervisor level** to prevent certain access types from happening. In particular, in the case of unpacking protection, **write access is disabled**. If a write is performed on a protected page, the monitoring starts as described in section 5.2.

5.4.2. Macro-Benchmarking

To assess the performance degradation on the execution of real-world applications, the macro-benchmark we developed simulates user interaction with the GUI of the monitored process. We designed one simulation flow for each process, personalizing the interaction to mimic a real-world scenario. Simulations were carried out both with monitoring and without monitoring the simulated process for code unpacking. The chosen applications are Microsoft Windows programs Paint, Notepad, Calculator, Task Manager, and Explorer. The first challenge associated with this family of benchmarks is the definition of a realistic simulation, in which ordinary tasks requiring user interaction are executed. To solve this problem, the following arbitrary sequences of actions have been chosen:

- *Paint*: upon Paint startup a .png file is opened, resized, and then saved to the disk

before closing the application.

- *Notepad*: the process is started with a blank untitled text document automatically created. The word "Test" is written in the document and it is saved in the home directory of the user under the name "example.txt".
- *Calculator*: the application is started, then the product of two numbers is computed and the app is terminated.
- *Explorer*: the folder "C:/Program Files/" is opened, a right click opens the context menu for the CommonFiles folder, and the Properties item is clicked. Last, the Properties window is closed and the Explorer.exe main window is terminated.
- *Task Manager*: the process is started, and a new task is created. The new task is an instance of "winver.exe", which prints information about the installed Windows version in a separate dialog. Finally, the "winver.exe" window is closed and so is the task manager one.

The second challenge associated with this benchmark is the implementation of user-like interaction automation. It has been solved by resorting to Windows GUI automation APIs for automatic graphical object manipulation in the context of the Windows graphical interface. A Python wrapper has been used to take control of GUI objects. The main complication to this kind of approach is related to time variability in GUI object instantiation. Empirical observations revealed that the GUI of the process and graphical object creation requires a variable amount of time for the correct displaying of all the elements. Such time amount is unknown and dependent on the overall operating system and hardware state (e.g. there could be scheduling latency) and on the hypervisor-based monitoring solution, which slows down GUI operations.

To cope with this problem, artificial sleeps have been introduced in the Python wrapper in points observed to be important for the overall simulation execution during the tests. Moreover, Windows GUI automation APIs allow the definition of timeout thresholds for managing the variable amount of time to wait before dynamically rendered graphical components are visible and ready to be used in the graphical interface. This facility has been extensively used to make GUI manipulation compatible with the aleatory overhead and delays introduced by hypervisor-based monitoring. Time collection happens inside the Python wrapper above described using the highest-precision time measurement tools. For each process, the start time is taken before the beginning of the GUI simulation and the end time at the very end, analogously to what has been done for section 5.4.1. The time taken by the simulation to run is the difference between the two measurements just

mentioned. Simulations are repeated a certain number of times for each process and results are averaged to get an average estimate of the time taken by each simulation. These steps are repeated both in the presence and in the absence of monitoring and then data are compared.

On the monitoring part, the approach used to synchronize with process creation and track memory operation extends the one adopted in section 5.4.1 to take into consideration the multiple pages allocated inside the virtual address space of the monitored process. Conceptually, each process page is protected with a **write hook** on its allocation and, as soon as program execution starts, if traps in the hypervisor are generated upon memory writes in the hooked pages, the monitoring flow starts. The main problem with the simplistic approach of section 5.4.1 is that in this case broader visibility is required. All the pages allocated to a process have to be protected and the analyzed processes are non-patched real-world ones. It is no longer possible to assume that hooking a system call and analyzing the parameters of its invocation will inform the hypervisor-based monitor that the process memory layout has been set up. In this case, kernel functions responsible for memory management and process creation are hooked to devise a method general enough to track dynamic memory accesses.

In the case of Drakvuf, an extension plugin called **Codemon** has been used to carry out the tests of interest. In Codemon, one routine responsible for handling page faults in Windows is hooked. This results in every possible page fault tracked, including the ones not relative to the memory of the monitored process. On the one hand, this introduces a considerable slowdown in the operating system overall since page faults are a very frequent event, and trapping in the hypervisor suspends the execution of the guest. On the other hand, every page access is effectively captured and, after some filtering in the VMI library, all the relevant pages are identified and hooked. First of all, this choice is mainly motivated by the need to have full visibility on the pages allocated without diving too deep into the details of Windows process creation and memory allocation. Secondly, Drakvuf is limited by design in the way hooks are installed by the library. It requires the hooked pages to be physically allocated at the moment of the installation of the hook. If pages have not yet been loaded or have been swapped out, the library cannot physically apply the modifications needed to hook memory accesses to such pages (to the corresponding virtual address).

Hooking one of the page fault handling routines in the OS, Codemon can synchronize with physical allocation and effectively install hooks before the OS or the process has access to the page but right after the page is in physical memory, potentially solving the problem. Last, but not least, Codemon has been in the Drakvuf codebase for a while, it

has been tested and progressively updated to handle correctly memory tracing, hopefully without creating instabilities in the guest while acting on low-level undocumented internal routines [34]. Codemon solves the semantic gap-related problem of figuring out which of the accessed pages are executable hooking indistinctly all the pages for **execute access**. The library learns that a page intended by the guest to be executable from the traps into the hypervisor caused by the **first instruction fetch** in that page and only after this event happens it installs the **write hook** as stated above.

We used the same hooking logic described in section 5.4.1 for monitoring the address space of a process against code unpacking with HVMI. The process to monitor is installed as a protected one in the library using its APIs, upon process start its entire virtual address space is protected against writing attempts, and, on the first write, monitoring logic starts as depicted in section 5.2. Like Drakvuf, HVMI can only install hooks on pages physically present in memory. However, one difference with Drakvuf is that HVMI does not need to track every page fault because it handles hooks differently. Upon hooking API invocation, the library installs hooks directly on the page table structures of the operating system. Page allocations in physical memory result in traps in the hypervisor because the OS pager traverses the page tables to install the page directory entries on page allocation or swap-in. This process is highly optimized in HVMI to limit at a bare minimum exits to the hypervisor exploiting some extended hardware features described in chapter 2 section 2.1.2. This means that, on process creation, **the executable portion** of the virtual address space of the process is hooked **against writes** by the library and, upon first access to a certain page by the process, a hook is installed right after being loaded into physical memory and, at the same time, before the actual access.

Based on these considerations, one core difference between the two approaches for tracking code unpacking in the context of a process resides in the way memory initialization by the operating system is handled. In the case of Drakvuf, the plugin we used in our approach puts on a page of interest the very first **execute hook** as soon as a page fault is generated at the very first access to the page. At that point of the guest execution, because of the semantic gap problem, the plugin has no way of knowing whether the accessed page is going to contain code or not. For this reason, the tool hooks all the pages indistinctly. On the one hand, because of how hooks are implemented, this produces **duplication of the hooked memory pages of the guest** (more on this in chapter 6). On the other hand, this allows precise execution tracing without the need to potentially resort to the header of the launched or to hooking process creation routines. As a plus, Drakvuf can monitor memory immaterially of permissions changes happening during guest execution. Each page being execute-hooked, if one page whose initial permissions set does

not include execute permission (X), and, subsequently, a program makes it executable to fetch instructions in it, the tool is capable of handling this case. On the very first execution attempt, the memory content of the page is hashed. This allows Drakvuf to track the content of monitored pages independently from relocations happening due to swapping activity. Despite hashing the page each time the guest traps in the hypervisor on execute violation potentially adds overhead, this prevents hooks from being installed multiple times on the same guest memory frame.

HVMI does not need to install hooks indistinctly on all the pages accessed by a process intercepting page faults. It uses data describing the memory layout of the program in the header of the executable associated with the process to be protected. This derives from the very high level of integration with Windows internal routines that HVMI has. It limits the number of traps in the hypervisor happening at process creation and during process runtime, as later explained in chapter 7, but it creates a limitation in the tool that Drakvuf does not suffer from. HVMI cannot track memory accesses to memory locations **dynamically allocated** at runtime by the process spawned from the executable set to be monitored. This limitation is exploited to show the inadequacy of HVMI in tracking code unpacking leveraging a modified version of the micro-benchmark program written to test the performance of memory hooking in chapter 7.

In our approach to assessing the overhead associated with process-wise hypervisor-based memory tracing, we deliberately ruled out the tracing of accesses to dynamically linked libraries. By default, Codemon can trace process accesses to shared code, while HVMI omits it and only statically traces dynamically linked libraries insertion in the address space of a process. To uniform the testing setting, Codemon has been patched to rule out DLL access tracing. The main motivation behind this choice is that when processes other than the one traced access DLL memory, traps in the hypervisor happen. The processes linking a DLL might access it very frequently across their execution. Since this leads to potentially many traps that are not related to tracing the operations of the monitored process, in our approach we **exclude DLL tracing**. The most important VMI tools do not effectively handle DLL tracing in a scalable way and the problem has been extensively investigated in the past. Solutions have already been designed and proposed [21], but they are not part of the most important and used VMI solutions yet at the time of the writing.

5.5. Taming the hypervisor-debugger

Although HyperDbg is a hypervisor, it has been designed for kernel and user-level debugging. To answer our secondary research question, we approached the task of retrofitting

HyperDbg for hypervisor-based monitoring focusing on its memory and register hooking functions. The hooking and tracing subsystem of HyperDbg was designed with efficiency in mind, and several optimizations allow it to be faster than all the state-of-the-art kernel level debuggers for Windows at installing and handling conditional breakpoints, hooking system calls and perform debugging step-in actions [31].

We tried to exploit the substantial improvement in the state-of-the-art in terms of debugging performance and the hooking capabilities of the tool for monitoring purposes. Monitoring with HyperDbg revolves around what the authors call an "event". An event may be a write to a given register, a read from a memory location, or an interrupt [31], and so on. Each event is associated with a callback, registered at the time of its activation. Under this point of view, Drakvuf, HVMI, and HyperDbg are similar, monitoring actions can be carried out upon event activation. Also, the same types of events can be captured by the three solutions [19, 23, 34].

It is imperative to stress one point, that has had a lot of influence on our approach to solving the code unpacking detection problem using HyperDbg: *HyperDbg is a debugger, not an introspection library.* The scripting engine of HyperDbg is not as powerful and flexible as the functions of HVMI and Drakvuf. Introspection capabilities are exposed by HyperDbg through its debugger CLI. Guests can be introspected leveraging the so-called VMI mode for the debugger, by which one process can introspect the operating system on which it runs from the user space. They are also available to an external host trying to debug a given guest through the facilities for remote debugging integrated into the hypervisor of HyperDbg. Nonetheless, the introspection capabilities of HyperDbg are limited. HyperDbg cannot flexibly manipulate hooks as done by Drakvuf and HVMI. By design, to improve the time needed to react to an event with a debugging action (setting a breakpoint, checking an address to see if it belongs to the process, and many more), the debugger implements most script execution stages in **VMX root mode**, i.e. in the hypervisor. Most actions do not require the guest operating system to be fully blocked and only stop the virtual CPU core where the violation happened. This is possible thanks to hardware support for per-vCPU core reconfigurability [5]. The hypervisor instantiates one VMCS structure for each vCPU core at boot time and, at runtime, this allows core-selective traps registration and debugging. All these design choices make event management fast and less invasive to guest execution [31], but they **pose limitations** to the functioning of the tool.

The most prominent one that we identified is the lack of flexibility in **trap management**. HyperDbg being a debugger, cannot switch memory hooks in response to EPT violations. Traps can neither be dynamically instantiated and installed nor removed.

Such a capability is not part of the simple script engine integrated into the tool because trap switching is not strictly needed for debugging purposes but it is at the base of our approach to memory monitoring and unpacking detection. This makes it impossible to replicate the same logic as Drakvuf and HVMI on HyperDbg. Exploring the design space to find an alternative approach to memory monitoring with HyperDbg has been the most difficult task to accomplish for answering the secondary research question. Nonetheless, we identified a strategy to **adapt HyperDbg to our memory monitoring use case**.

We explored the design space passing through all the hooking facilities, focusing on stealth SLAT-based hooking primitives. The priority was to implement a monitoring flow functionally equivalent to the one described in section 5.2. The most suitable feature to do so is EPT-based **debug registers emulation**. Debug registers are an important tool available at the hardware level in modern processors to provide fine-grained memory access tracing support for debug breakpoint and watchpoint implementation. Since they are limited in numbers in real hardware, the authors of HyperDbg made them available in an unlimited number in the debugger using EPT hooks for memory access control[31]. We leverage this feature for monitoring tracking write and execute accesses together, i.e. with only one hook per page statically installed without the possibility of disabling, destroying, and recreating it at runtime. Upon write access to the page, this latter is marked as "dirty" setting to one a memory location reserved inside one preallocated memory location in the hypervisor. On execute access, if the just mentioned memory location representing the state of the page is set to the value one, we assume unpacking is detected as described in the model in section 5.1. Otherwise, execute access is ignored.

Distinguishing a write access from an execute one has been the hardest challenge to cope with. To solve it, we used the following heuristic:

- If in the context of a trap, the accessed address where the violation has been generated is the same as the current instruction pointer of the suspended guest, we assume the access is an execution attempt.
- Otherwise, if the address of the violation and the instruction pointer differ, we consider such violation a write attempt.

It is important to stress that a corner case of this heuristic is the one represented by an instruction overwriting itself. With our method, such write would be exchanged for an execution. To cover such a corner case, an alternative way of detecting a write is needed. We assume that this event does not happen in our setup and it is of no interest to our case study since we want to approximate the behavior of the already presented introspection solutions, assess the capabilities of HyperDbg, understand its limitations, and estimate

the overhead resulting from its use.

Apart from the corner case, we prove with our experiments that the presented logic approximates from a functional standpoint the approach designed for the introspection solutions and that this hypervisor-based debugger is also usable as a memory monitor from a functional standpoint. However, as stated above, our experiments also showcase the limitations that make it impossible to approximate functionally Drakvuf and HVMI monitoring while **retaining efficiency**. In particular, with no dynamic trap management available, the number of exits to the virtual machine monitor that are not necessary for monitoring to detect unpacking becomes high. We want to prove that this is not a limitation in the approach, but in the way the tool is built.
To carry out tests on HyperDbg, we employ a modified version of the micro-benchmark proposed in section 5.4.1. In this version, the code that is going to be executed and overwritten in a loop is inside a memory chunk of the same size as a page allocated using Windows dynamic memory allocation APIs, i.e. 4096 bytes. At the beginning of the benchmark, the debugger attaches to the monitored process, extracts the address of the page of interest, and installs the hook on it. Upon the first write, the monitoring logic depicted in section 5.2 starts. Putting the monitored code in a different memory region from the one where the main is allocated allows us to limit the noise in the time measurements collected. In this way we quantify the overhead associated with monitoring for unpacking detection with HyperDbg memory hooks and no spurious write and execute accesses get in the way, i.e. only the events of interest are captured. This test bench supports us in proving that trap-handling for memory monitoring is very efficient in HyperDbg, even more efficient than introspection libraries.

Nonetheless, as far as macro-benchmarking is concerned, testing HyperDbg was made **impossible** by the way it is implemented and its lack of dynamic trap management. Running HyperDbg-based monitoring on a real-world application would mean that an exit to the hypervisor happens at each memory write/execute inside a monitored page. If an exit happens at each instruction fetch, the execution of the guest is single-stepped. This is perfectly fine in the context of debugging a program closely following internal state transitions, as is done in practice with commodity debuggers. However, it does not fit the use case of hypervisor-based monitoring, as efficient monitoring assumes traps in the hypervisor are kept to a bare necessary minimum.

For this reason, we claim that using HyperDbg for memory tracing of applications that are more complex than the simplistic presented micro-benchmark **is infeasible**. We believe that the frequency of the traps experienced when testing such applications would be so high that guest execution would be dramatically slowed down. And we believe that it will

be even higher when monitoring more complex applications.

To prove this assertion, we made further modifications to the HyperDbg micro-benchmark to allow lengths of the shellcode greater than one to be used. At startup, the process takes as input the desired length of the shellcode to be executed inside the $\mathbb{W} + \mathbb{X}$ loop. Such length is the number of bytes that the shellcode is made up of. Upon parameter parsing, the benchmark dynamically extends the code to be executed in the above-mentioned loop including as many NOP instructions in the prologue of the script as bytes of the length specified by the user in input minus one, because the last byte is a return (RET) instruction. In x86, the NOP instruction is just a non-state-changing instruction used for various purposes, such as implementing busy-waits [5]. We choose NOP because it is a one-byte instruction and it eases the task of proving that the overhead incurred by the monitored program grows with the length of the executed (and monitored) shellcode. Many tests are carried out adding NOP instructions at each iteration. Apart from the parametric shellcode extension just described, the experiment follows the same logic as the micro-benchmark described for the HyperDbg use case. Time was collected in the same manner as well.

In our opinion, this approach to dissecting and analyzing the performance overhead of HyperDbg leads to proving that it cannot be used when monitoring more complex applications than the micro-benchmark without incurring a potentially prohibitive performance penalty.

6 | Implementation details

In this chapter, we delve deep into the implementation details of the studied solutions and of the proposed tests and extensions.

6.1. Architectures Overview

This section introduces the high-level software architectures for the introspection tools that have been investigated in the present thesis.

6.1.1. Drakvuf

As presented in chapter 2, Drakvuf is a virtual machine introspection architecture running on top of the Xen hypervisor. Xen is a type 1 hypervisor implemented as an exo-kernel, i.e., a simple monolithic software layer for implementing coarse-grained hardware multiplexing [58]. The virtual machines it instantiates and manages are called Domains. In the setup used to carry out Drakvuf performance evaluation Xen is configured and managed through the use of a special virtual machine automatically instantiated at boot time, called Dom0. Xen places some constraints on the operating systems usable as Dom0, in that they have to be a Linux-based distribution with paravirtualization support. Paravirtualization is a technique implemented to speed up virtualized operating systems hardware-related operations through the use of ad hoc drivers which are hypervisor-aware. The other instantiated virtual machines are denoted with the expression DomU, where the "U" stands for whatever domain number greater than 0. Those are the guests in the Xen virtualization system and supported operating systems also include Linux-based ones and every Windows version. Drakvuf is a userspace library and, in our setup, it runs in Domain 0. It requires privileged access to the hypervisor to control its functionality and install hooks in the guest. For this purpose, Drakvuf runs with root privileges on Dom0 and controls Xen using the libxenctl library. Communication with hypervisors revolves around the concept of hypercall, a type of cross-domain call initiated by a virtual machine to invoke some functionality provided by the hypervisor. Libxenctl is a wrapper around hypercalls providing a suitable level of abstraction to account for the variability of the

Xen hypercall subsystem on different platforms. The libxenctl library is used to create, destroy, and control the virtual machines. Moreover, Drakvuf is an extension of the LibVMI introspection library. An instance of LibVMI is created at Drakvuf startup and it runs in userspace as well. Introspection actions put in place by the two libraries revolve around the concept of Xen Event. In Xen, an Event is a communication abstraction that represents a change in the state of the hypervisor or of the virtual machine or signals the execution of a particular action of interest in one of the Xen guests. Events are dynamically set up to be generated on arbitrary conditions (e.g. guest accesses to a protected memory location). There is one event channel per guest which is accessed exclusively by the instance of the LibVMI library created at Drakvuf startup. Responses to events are passed to Xen using the libxenctl APIs and they are how active introspection is implemented [34, 44]. Figure 6.1 describes on a high level of abstraction the software architecture implemented to evaluate Drakvuf.

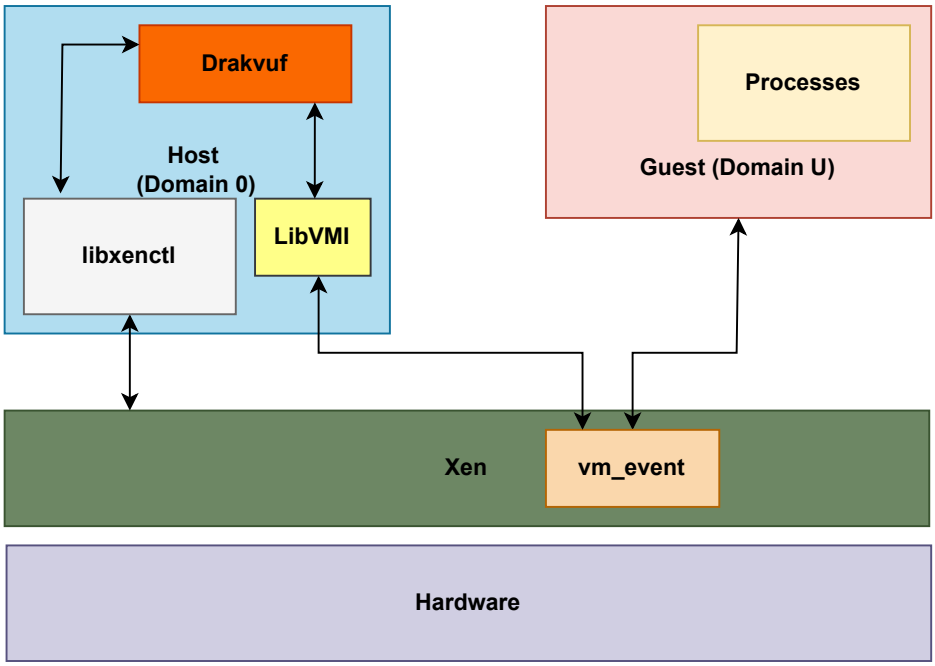

Figure 6.1: High-level architecture of Drakvuf on Xen.

6.1.2. Napoca

Napoca is a type 1 hypervisor like Xen. However, only one virtual machine can be instantiated and managed by Napoca. While Xen is concerned with scalable and efficient management of multiple guests on the same hardware, Napoca only works to provide a high-privileged software environment for forensics activity on a running operating system instance. The only supported operating system is Windows 10. As suggested above,

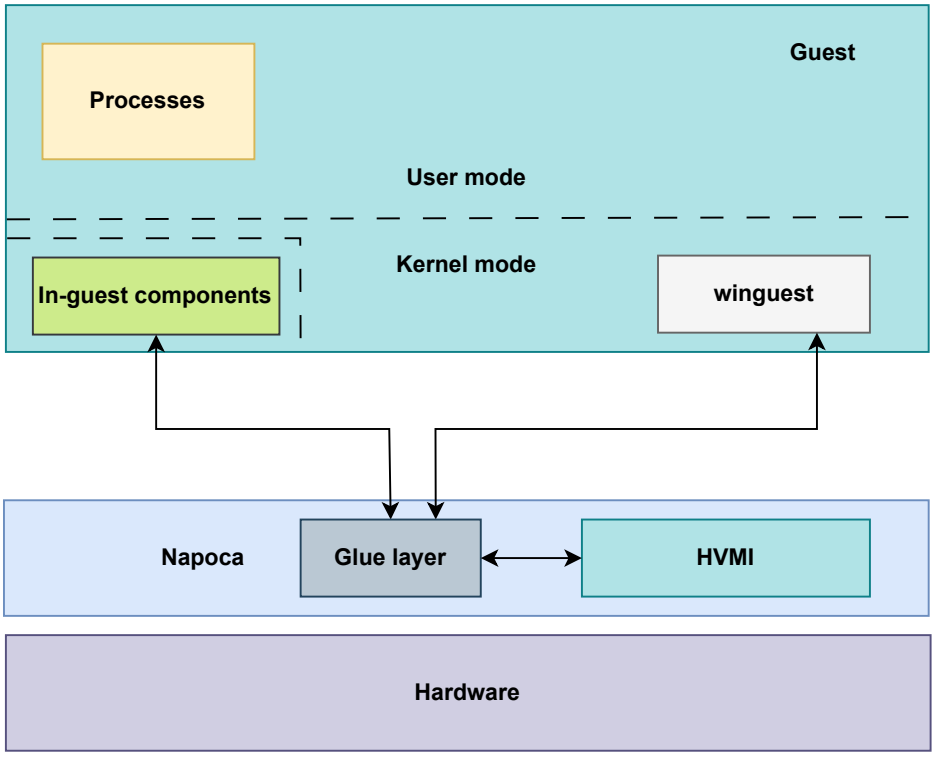

Figure 6.2: High-level architecture of HVMI on Napoca.

Napoca has been designed to ease the integration of introspection solutions directly at the hypervisor level for efficiency. Its architecture is monolithic, no modules can be installed to extend its functionality except from an introspection library for fine-grained guest execution control. To support such kind of extension, Napoca internally exposes functions aimed at supporting tracing and introspection as depicted in chapter 2 section 2.1.3. They are internally exposed in the sense that they are not directly accessible from the guest, but they are callable from the introspection library. To have the introspection library running at the hypervisor level, the statically linked library is placed in a predefined disk location. The hypervisor loads it into some preallocated memory pools at boot time. Communication between the library and the hypervisor relies on a "glue layer", a shared interface between the hypervisor and the introspection library. The library we evaluated is called HVMI and Napoca natively supports it thanks to the glue layer provided together with the hypervisor by the authors. Different from Xen, where event channels are used to implement inter-guest communication and active introspection, Napoca relies on the concept of callback to enable introspection library support. The library registers introspection callbacks upon initialization or dynamically in case of dynamic reconfiguration at runtime. The hypervisor invokes the registered callbacks when the conditions for their execution are met. This mechanism is used to implement active introspection [19].

Figure 6.2 describes on a high level of abstraction the software architecture implemented to evaluate Napoca.

6.1.3. HyperDbg

HyperDbg is an introspection-based debugger integrated into a type 1 hypervisor. The hypervisor is loaded on-demand at runtime by a Windows kernel driver which virtualizes an already running operating system instance. This is one of the main differences between HyperDbg and the already presented hypervisors. In particular, in our setup, HyperDbg is installed into a virtual machine already running on top of the VMWare Workstation type 2 hypervisor and the debugger is controlled using the emulated serial interface provided to the host operating system by the VMware Workstation hypervisor. This choice is warmly recommended by the developers of HyperDbg since it is not production-ready and crashes happen frequently in an experimental setup. Running the hypervisor debugger in a virtualized context prevents such crashes from hampering the correct functioning of a non-virtualized host. HyperDbg is installed in the debuggee right after its boot in our setup. The integrated hypervisor virtualizes on-the-fly a running operating system instance. Since that is done on an operating system instance running in a virtualized context, we talk about nested virtualization. In nested virtualization, a hypervisor is installed on top of an already running one leveraging Intel hardware support for efficiency and practicality. In our case, the VMWare Hypervisor emulates VT-x and makes them available to HyperDbg. This latter installs virtualization support passing through a driver which loads a VMM module to create a nested VMX non-root mode where the monitored guest is scheduled to run. The installed VMM runs in nested VMX root mode and has full control over the exits to the monitor performed by the doubly virtualized guest.

In our setup, the process of triggering the installation of HyperDbg is initiated by the debugger controller. The debugger controller is a console app running in the host operating system that manages HyperDbg installation in the debuggee with the help of another instance of such controller running in the context of the virtualized operating system to be monitored. As stated above, communication between the debugger and the debuggee controllers is implemented using the emulated VMware serial interface. A serial interface is a communication facility integrated into the motherboard of some machines to provide low-level control over the execution of the machine, usually for debugging purposes, as also happens in Napoca. Such ports come physically in many formats such as RS-232, for example.

VMware emulates such a hardware interface using process pipes. Process pipes provide

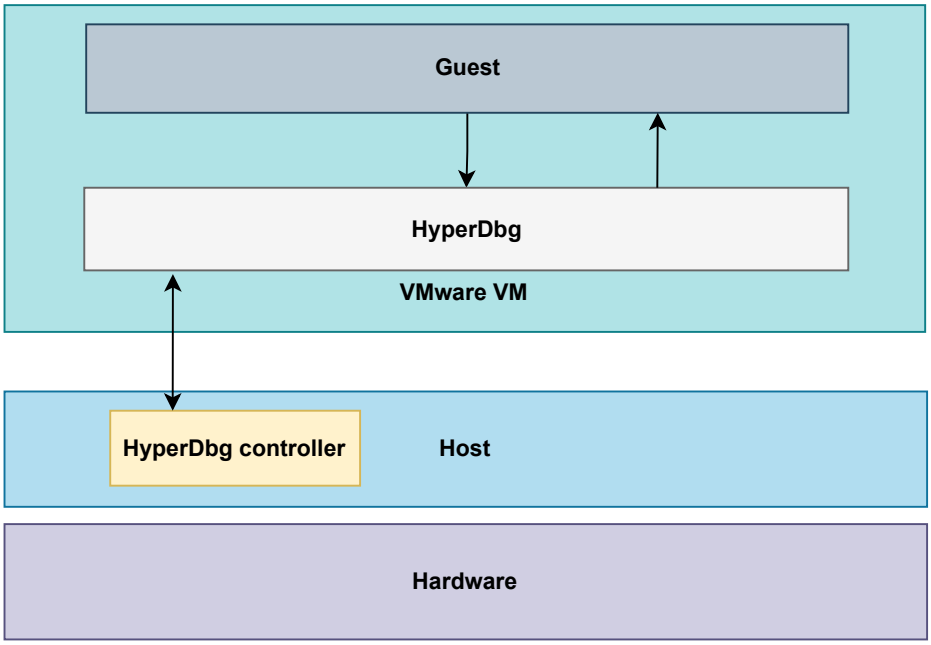

Figure 6.3: High-level architecture of HyperDbg.

support for buffered interprocess communication on most commodity operating systems. A named pipe, i.e., a pipe exposed to processes by the OS and identified using a name defined at creation time, is used to send debugger commands (e.g. for setting breakpoints, watchpoints, disassembling memory locations, etc.) and settings to the HyperDbg hypervisor in the virtual machine. Sent commands are executed in the context of the debuggee hypervisor and command execution results are sent to the debugger controller in the host using the same pipe as above. Events generated by the guest, such as EPT violations and writes to previously protected registers, may either follow the same iter along the pipe or may just be handled upon suitable configuration in the HyperDbg hypervisor [4]. Figure 6.3 provides a high-level perspective of the HyperDbg architecture used in the present thesis.

6.2. Micro-benchmark

The logic at the base of the micro-benchmark is shown in Algorithm 6.1.

It has been implemented as a C++ program using Windows memory management APIs. Two versions of this benchmark have been designed: base and custom. The base version has been used to carry out tests on Drakvuf and HVMI. The custom version, instead, has been adapted to the methodology used for HyperDbg. The following describes the implementation details related to the base version. The main file contains the declaration

Algorithm 6.1 Algorithm describing the flow of the micro-benchmark.

```
1: shellcode ← {RET}
2: p ← ptr_to_code_page
3: make_rwx(p)
4: copy_shellcode(shellcode, p)
5: iterations ← arg1
6: start_time ← time()
7: for i in range(N) do
8:     write_at(p)
9:     fetch_instruction_from(p)
10: end for
11: end_time ← time()
12: output(end_time - start_time)
```

of two routines, *main* and *overwrite_me*. The main function implements the algorithm above. At its beginning, an array of char objects is instantiated and initialized to contain some shellcode. That is going to be written to a memory location which, in turn, is going to be run and overwritten in the loop. In our micro-benchmark, the shellcode is minimalistic and only consists of one byte, whose value is `0xc3`, which is the processor opcode for the RET instruction. Such an instruction is used to return from a procedure call. The memory location that is going to be overwritten is included in one of the executable pages allocated to the process. In particular, it is the prologue of the *overwrite_me* function. After allocating the shellcode buffer, the benchmark grabs a handle to the just cited routine *overwrite_me* and changes the permissions of the memory frame in which it is loaded at runtime to make it writable. This is done using the *VirtualProtect* Windows API, specifying as parameters, in order, the virtual address of the page, the size of the area whose permissions are to be changed, the new permissions set (Read, Write, and eXecute), and a pointer to a DWORD (signed 32-bit integer) where to store the old permissions of the page.

This operation is needed because executable code memory pages are not usually writable in modern operating systems for security reasons [54]. Upon the change of permissions, the program instantiates a pointer to a character array referencing the memory location of the *overwrite_me* function. This allows the program to write at the byte granularity on the page since in C++ on the Windows Visual Studio Compiler (MVSC) the type char maps to a signed 8-bit integer, i.e., a byte. The first for loop in the program writes the shellcode to the address of the *overwrite_me* function. At this point, the number of iterations to be performed by the subsequent loop, already mentioned in chapter 5 is either parsed and converted into an integer value from user input or fixed to a conventional value of 100000. The most relevant part of the micro-benchmark starts with the collection

of one time measurement. After this, the $\mathbb{W} + \mathbb{X}$ loop starts, and the initial byte of the prologue of the *overwrite_me* function is executed with the call to the function, just before being overwritten. At the end of the loop, another time measurement is taken and the duration of the loop is computed by subtracting the first measurement from the second one. The duration is then printed to the standard output. Measurements are taken using the C++ high-resolution clock, the clock with the greatest resolution among the ones provided by the language runtime. The program also includes useful debug prints and variable definitions for convenience. The source code can be found in appendix A.

In the case of HyperDbg, the program has been adapted to include the allocation of a memory chunk of the same size as an ordinary memory page, i.e., `4096` bytes, using the *VirtualAlloc* Windows API for dynamic memory allocation. The API is invoked with the `MEM_RESERVE` and `MEM_COMMIT` flags to reserve and commit the memory area. In Windows userspace memory allocation APIs, committing a memory interval means informing the operating system that the program is expanding its virtual address space. The operating system, in turn, checks whether enough resources are available to satisfy the request. Memory is then reserved to make sure that such memory location is not used by other dynamic allocation functions in the context of the process [38]. The modified version of the micro-benchmark no longer grabs a handle to the code page containing the *overwrite_me* function, which is not even present in the code, since it is going to use the newly allocated area to write the shellcode to be executed. After its initialization with the shellcode, the program changes the permissions of the newly allocated page to make it executable. Upon the invocation of the *VirtualProtect* function mentioned above, the program execution continues as in the base version.

6.2.1. Monitoring with Drakvuf

Drakvuf is based on altp2m, as explained in chapter 2. Before delving into the details of the library per se, we wish to investigate the overhead resulting from the use of page table switching at the hypervisor level for memory monitoring.

EPT views can be switched using the already mentioned libxenctl. To carry out the test on the performance overhead resulting from the switch, we wrote a C program that invokes the APIs of such a library. First, the program instantiates a handle to libxenctl, then asks the hypervisor to enable altp2m, disabled by default on a just-created Xen domain. Subsequently, a new view is created.

Command in Listing 6.1 is used to create a new view, whose index is saved in the `newlyallocv` variable. The permissions set of the new view is identical to the princi-

```
int rc = xc_altp2m_create_view(xch,
                               domid,
                               XENMEM_access_default,
                               &newlyallocv)
```

Listing 6.1: Altp2m view creation.

pal view associated with the current guest with identifier `domid`. The variable `xch` is the handle to the hypervisor control library. After that, the program switches the domain to that view and then switches it back to the original one. In the end, we deallocate the view and the instance of libxenctl.

```
    // switch to the new altp2m view
if (xc_altp2m_switch_to_view(xch, domid, newlyallocv) < 0)
{
    fprintf(stderr, "Error switching to the new view
                        with ID: %d\n", newlyallocv);
    // error handling, omitted here ...
    goto exit;
}
```

Listing 6.2: EPT View switch on Xen.

In the hypervisor context, a routine is hooked to take time samples at the beginning and at the end of the invocation of the function that switches the domain to the new view. This latter function is *p2m_switch_domain_altp2m_by_id*, defined in *xen/xen/arch/x86/mm/p2m.c*.

```
int p2m_switch_domain_altp2m_by_id(struct domain *d, unsigned int idx)
```

Listing 6.3: Internal routine to switch page table pointer on a Xen VM.

Its behavior is hereby summarized: the hypervisor stops the domain, acquires a lock over the per-domain page table views list, checks if the provided index `idx` is valid for the domain, and overwrites the `EPTP` inside the respective `VMCSs`. In the end, internal data structures are updated and the domain is restarted under the new view. The time samples are printed in the Xen console logs. Results are collected automatically using a script that periodically flushes the log buffer of Xen to a file. The file is then processed to extract timing measurements and the average time is computed over 500000 samples. The hooked routine is in Listing 6.4 and it is the routine in charge of handling altp2m-related hypercalls.

```
static int do_altp2m_op(XEN_GUEST_HANDLE_PARAM(void) arg)
```

Listing 6.4: Xen internal routine to handle altp2m operations.

It is declared in the file xen/xen/arch/x86/hvm/hvm.c, and it has been modified as shown in Listing 6.5.

```
/* in an enormous switch for hypercall opcode filtering*/
 case HVMOP_altp2m_switch_p2m:
    int64_t s = NOW(); // added code
    rc = p2m_switch_domain_altp2m_by_id(d, a.u.view.view);
    int64_t e = NOW(); // added code
    printk(KERN_INFO "altp2m switch: %ld ns\n", e-s); // added code
    break;
    /* the routine continues */
```

Listing 6.5: Point of the Xen codebase where p2m_switch_domain_altp2m_by_id is invoked.

The patch above logs to Xen buffer the time taken to switch the domain to the new view. The time is measured using the *NOW()* macro, which is a wrapper around a Xen utility used to read System Time. Results can be obtained by filtering the logs as in Listing 6.6.

```
  xl dmesg | grep "altp2m switch" > altp2m_switch.log
```

Listing 6.6: Altp2m test results extraction.

Monitoring using Drakvuf on top of the Xen hypervisor has been implemented extending the syscalls plugin. This plugin implements VMI-based system call invocation and return value tracking, both on Windows and Linux and both on 32 and 64-bit architectures. At initialization, a file path has to be provided to the plugin. It contains the list of system call names representing the system calls to be hooked. At initialization, system calls are located by reading from the kernel memory the value of the variable "KeServiceDescriptorTable". It points to the SSDT (System Service Descriptor Table), the data structure used to translate a system call number into the associated kernel level handler routine address [29]. The plugin accesses the addresses of the routines traversing the table and, if the routine has been included in the configuration file provided at startup by the user, a stealthy EPT-based breakpoint hook is installed on the page leveraging libdrakvuf APIs. To install such a breakpoint, Drakvuf creates an executable mapping of the page containing the address to be hooked and overwrites the byte at the address where the hook is to be placed with the `0xcc` byte, which in x86 raises software interrupt number 3, i.e., a debugger interrupt. The stealthiness of the hook is guaranteed by the use of **altp2m** on Xen as explained in chapter 2.

During the library initialization phase, Drakvuf invokes libxenctl to create three EPT views starting from the original one for the monitored virtual machine. For each view, a different set of permissions is allowed on page entries present in the view. The most

used view is deemed `idx` in Drakvuf. This is the view used for mapping hooked pages to modified shadow pages. A shadow page is a memory page where breakpoints and other modifications to the execution flow are written. Instructions that are at a monitored guest physical address are fetched from the relative shadow page to implement hypervisor-backed guest execution flow control. For this reason, it is the primary view used by the guest during its execution.

The view `idr` maps the modified pages to a sink page only containing bytes with the value 0xff. A read to a shadow page physical address is redirected to the sink page and the returned value is a 0xff byte sequence made. The last view is called `idrx` and it maps the shadow pages of a restricted monitored context to the sink page whenever the guest is not running in that specific context. Here, the word "context" can be associated with a memory hierarchy, i.e., a CR3 context or, equivalently, either the context of a user process or the context of the kernel. Its usage and importance will become clear in the following.

Through such memory views, Drakvuf exploits altp2m to exert fine-grained execution control. Instead of relaxing the permissions of a page where a violation occurred, as HyperDbg and HVMI do, Drakvuf restarts the guest after switching the EPT view to the one with the right permissions set to carry out the specific action that generated the violation and the in-guest offending instruction is reexecuted[33].

Going back to the specific case of EPT-based stealth breakpoints, if, for example, a breakpoint is placed at guest physical address `0x12345678`, the page containing such an address is mapped to a shadow page in the `idx` view. If the guest hits the shadow mapping of such a physical address, a trap in the hypervisor occurs. Drakvuf handles the trap invoking the callback associated with it at the time of the registration of the breakpoint and restarts guest execution using the guest EPT view installed by the hypervisor at guest boot time.

Before restarting the guest, single-stepping is activated setting MTF. The guest is already set to trap in the hypervisor on debugging exception, so, after the execution of the instruction who trapped in the first place, a new trap happens that is managed by Drakvuf restoring the `idx` view. In this way, the guest restarts running instructions fetched from the special `idx` view. A very similar way of handling trap is used for read and write violations happening on the `idx` view while the guest is executing. Read and write attempts are considered legitimate if the initial guest-controlled permissions of the page allow for them.

In such cases, either read and write operations are emulated using the `idr` view (if executed on guest addresses that map to the machine addresses of shadow pages, which are

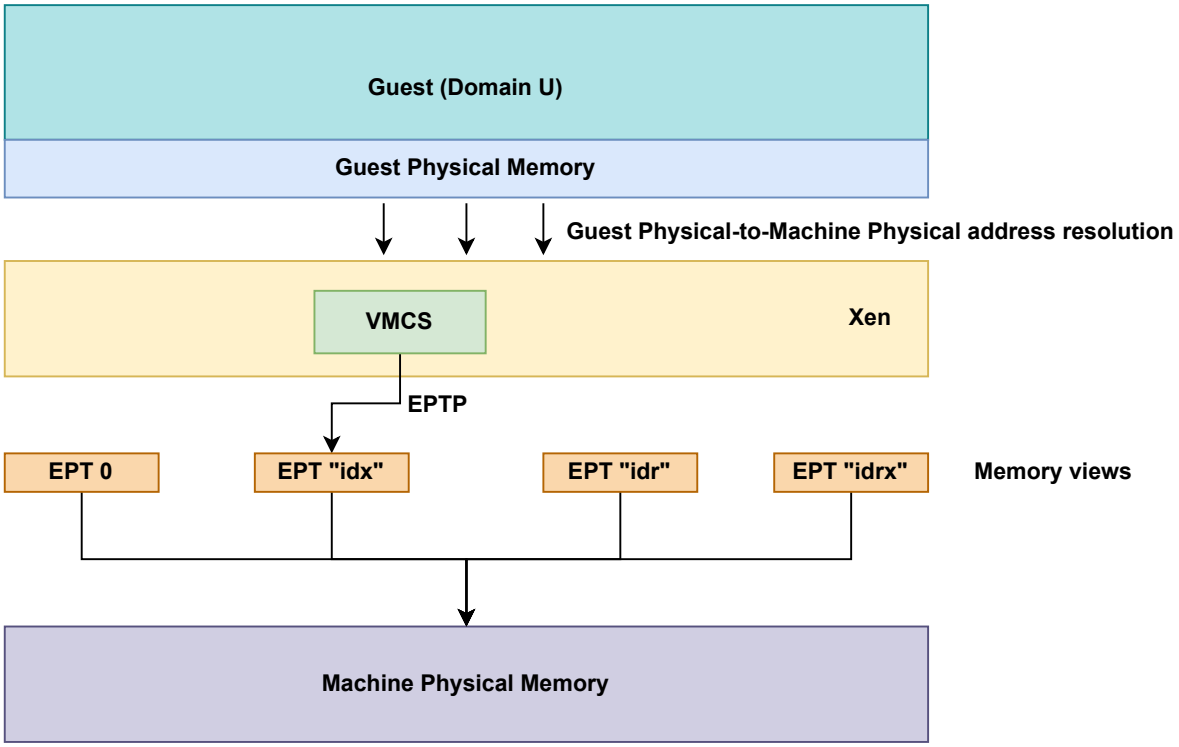

Figure 6.4: Altp2m high level overview.

protected by the hypervisor) or they are executed upon a view switch to the initial guest EPT view. Even in this case, single-stepping is required to restore the `idx` view after the execution of the offending instruction. In this context, emulating means returning a different value from the one written inside the page, that is taken from a sink page.

Implementing stealthy breakpoints with altp2m involves **duplicating** the machine physical page that the guest address where the breakpoint was installed maps into. At breakpoint installation, one copy of the whole page is created. The `idx` view maps the break-ed address into this copy. Such a page is modified to write into memory the software breakpoint opcode. The original version, without the opcode, is used to restart the execution of the guest after it breaks into the hypervisor at the breakpoint hit [1]. Figure 6.4 shows a high-level view of Xen altp2m-based EPTP switching. Figure 6.5 exemplifies the translation steps for going from a guest virtual address to a machine physical one in the case of altp2m: each view remaps the same guest physical address to a different machine physical one.

The file provided to Drakvuf at startup, in our very specific case, contains only the name of the NtProtectVirtualMemory system call. Such routine is an undocumented Windows internal system call handler called upon the invocation of the *VirtualProtect* function in

[1]Drakvuf source code: https://github.com/tklengyel/drakvuf

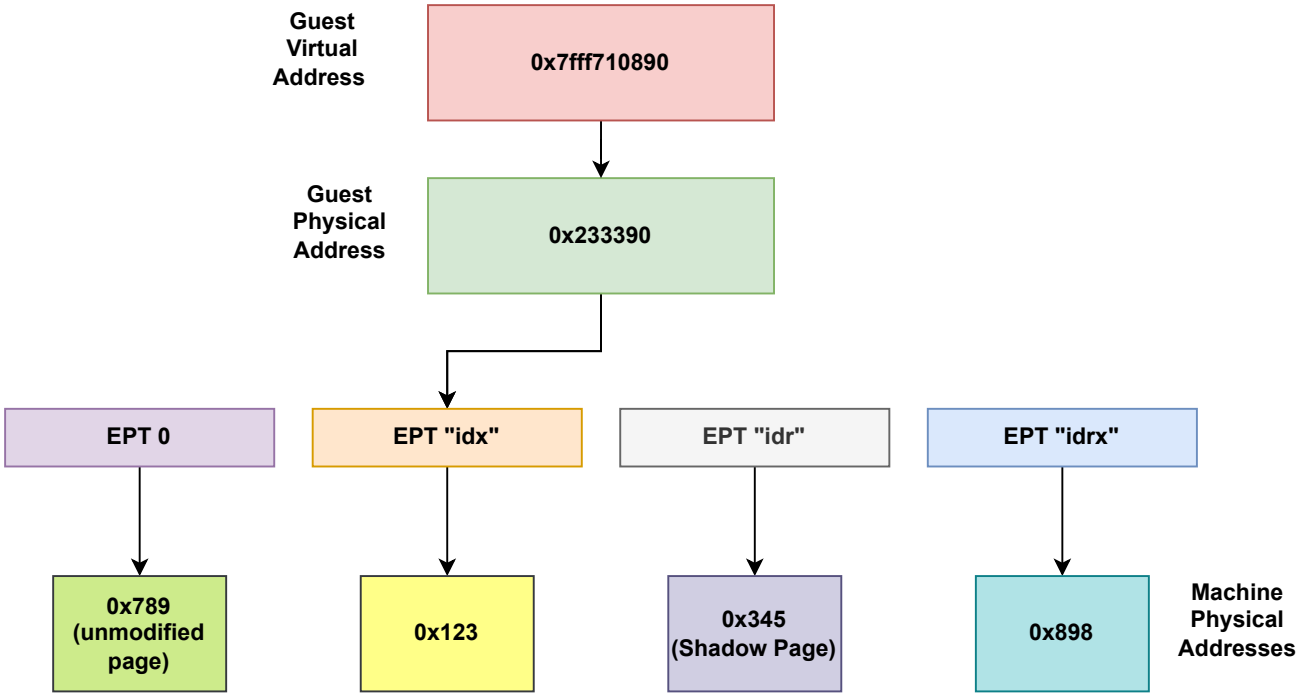

Figure 6.5: Different guest physical to machine physical address mappings used by altp2m.

the NTDLL library, which is the routine invoked in the micro-benchmark to change the permissions of the page to be monitored. The callback associated with the installed hook activates the trapping logic. Every time the Drakvuf monitor is started as reported in Listing 6.2.1. The starting command sets a context-sensitive view on the process to be monitored and specifies both the plugin to be enabled (Syscalls) and the file containing the list of system calls to be hooked. Since Drakvuf can also install hooks on the kernel routine responsible for returning to user space, but we are not interested in catching the return value of the system call, the command invocation sets the plugin to discard sysret hooks. At last, the name of the monitored Xen domain and a configuration file summarizing debugging data relative to the guest kernel are passed to Drakvuf.

```
sudo ./drakvuf -d win10 -r win10.json -a syscalls  \
                    --syscall-hooks-list ./syscalls_to_hook  \
                    -C --context-process unpacker.exe --disable-sysret
```

Listing 6.7: Prompt for starting Drakvuf for the micro-benchmark test.

To clarify, a context-sensitive view is an abstraction used by Drakvuf to only restrict the monitoring to a single execution context. The chosen context has its memory remapped inside the `idrx` view, used to run other processes in a different context without them being able to spot modifications to the memory layout performed by Drakvuf. The monitored process, instead, runs in the `idx` view, as ordinarily happens in Drakvuf-monitored virtual

machines. Whenever a hook is installed in the process, the interested memory location is remapped as stated above, and, since there is the need to hide modifications from other processes, their write, read, and execute attempts over guest physical location belonging to the monitored context fall back to the `idrx` view.

Context-sensitive monitoring has been empirically observed to lower the overhead of the introspection on the guest since a single process is monitored and hooked. For this reason, we chose to isolate the monitored process in a separate context to effectively measure the overhead on the single process resulting from the monitoring with as little noise on the estimates as possible. After the execution of the command, the micro-benchmark program can be run. In context-sensitive mode, Drakvuf installs a write hook on the **CR3 register** to synchronize with the scheduler of the guest as specified in chapter 2. If a process whose name matches the one specified as a parameter at startup is scheduled to run, the memory view on which it executes is set to `idx` and instructions are fetched from the modified/hooked shadow pages. This means that Drakvuf automatically detects the switch to the context to be monitored at runtime.

When the hooked system call is executed in the monitored context, a custom callback inside Drakvuf is activated. The routine is a patched version of the pre-existing *syscall_ cb* in the syscalls plugin. The original routine tracks system call invocation parameters, logs them, and, optionally, puts a return hook in the guest to track the return value of the system call. In our patch we still extract system call invocation values, starting from the new page permissions.

Listing 6.2.1 reports a description of the signature of *NtProtectVirtualMemory* taken from [9].

```
1 NTSYS\acrshort{api} NTSTATUS NT\acrshort{api}  NtProtectVirtualMemory(
2           IN HANDLE               ProcessHandle,
3           IN OUT PVOID            *BaseAddress,
4           IN OUT PULONG           NumberOfBytesToProtect,
5           IN ULONG                NewAccessProtection,
6           OUT PULONG              OldAccessProtection
7         );
```

Listing 6.8: Signature of the NtProtectVirtualMemory function.

The new page permissions are described by the "NewAccessProtection" variable. If permissions include both execute and write, we assume we found the virtual address of the page to be monitored. This assumption holds since, from empirical observations, the routine is only invoked at process creation and destruction, but not in other locations of the benchmark main body. The second step is extracting the guest virtual address of the

page to be monitored. This is done by reading from the user mode process stack the value of the virtual address using the *vmi_read_64_va* function. This function takes as input a handle to an instance of LibVMI grabbed using the *vmi_lock_guard* macro, the virtual address to read from, the pid of the process of interest, and a pointer to a variable where to store the read bytes. The address is the *BaseAddress* parameter mentioned above. Upon fetching this latter guest virtual address, internal LibVMI caches are flushed to be sure that no stale data are returned by the library.

Subsequently, the function *vmi_pagetable_lookup_extended* is called. The first two parameters this function takes as input are the handle to the LibVMI instance and the directory table base (DTB) of the monitored process, i.e., the base pointer to the page table directory of the process. This latter is stored in the CR3 register of the guest when the trap to the hypervisor happens after hitting the stealthy breakpoint. The second two parameters are the virtual address of the page to be monitored and a pointer to a variable where to store the guest physical page address. When mapped into memory, pages are referred to as frames. The value of the Guest Frame Number which identifies the memory frame to which the page is mapped is computed discarding the last 12 bits of the guest physical address [2]. Upon its extraction, the GFN is passed as input to the *createMemAccessHook* function to install a hook on it, registering the function *write_cb* as a callback and specifying that the hook has to fire before any guest write accesses to the frame. Such hooking API is exposed by the libhook library, a wrapper for implementing safe memory management for the hooks in C++. This library instantiates hooks using the RAII (resource acquisition is initialization) paradigm. In brief, thanks to RAII enforcement, when objects go out of scope, they are automatically garbage collected by the language runtime. Finally, a reference to the hook is moved into the syscalls plugin object using C++ move semantics for reference passing among different contexts to prevent pre-mature hook deallocation.

Upon write access to the monitored memory location, the *write_cb* fires. The write hook is reset and an execution hook is instantiated using the same API as above. Execute accesses are managed by invoking *execute_cb*. The routine handles the trap reinstalling the write hook, after incrementing the "trap_counter" variable. As the name suggests, this variable keeps track of the number of execute accesses happening to the monitored page when it is in a "dirty" state, i.e., after it has been written and the write hook has been swapped with an execute one. On termination, the Syscalls plugin is deallocated invoking its destructor, which takes care of logging the number of times the execution callback fired. This has supported testing activity since it allowed us to check that the hook has

[2]Libvmi source code: https://github.com/libvmi/libvmi

been activated as many times as the iterations of the $\mathbb{W} + \mathbb{X}$ loop in the micro-benchmark.

6.2.2. Monitoring with HVMI

HVMI implements fine-grained user space monitoring actions on top of the Introcore VMI library. As explained in chapter 3, EPT hooking is realized by modifying page table entries permission and installing callbacks that may either run in the hypervisor or in the host to handle violations. Only one EPT view is used when optimizations are not enabled and they are managed by an internal routine called *IntHandleEptViolation* [16]. Such routine may block memory accesses, emulate reads returning the original value of a page or a modified one in case of memory cloaking, or ignore writes to protected memory locations whose content has to be preserved. If the access is legitimate, the guest will be restarted to carry out the offending instruction. To restart the guest after handling a violation, the principal mechanism the library uses is based on activating the MTF. The guest is resumed with a relaxed set of permissions for the page which generated the violation and then single-stepped due to the MTF. Trapping in the hypervisor on breakpoint exception, HVMI restores the permissions set installed for hooking the page. In our setup, the violations of interest for the monitoring, on a very high level of abstraction, follow this flow.

In our case study, the library runs directly **in** the hypervisor. Upon suitable configuration through the boot hypervisor command line, it loads HVMI into its memory and jumps at the entry routine to initialize monitoring infrastructure. Protection measures are installed in two steps. The first step consists of enabling system-wide user-space protection through HVMI APIs.

The second step is dynamically activating ad hoc protection measures for the processes that we want to harden using the hypervisor. We modified the implementation of this last part to install on a process we want to monitor the same **trap-switching** logic presented in the case of the Syscalls plugin on Drakvuf. This time we do not do so by hooking a system call. Instead, we reutilize and patch the already existing Introcore facilities to help detect unpacking. In the case of HVMI over Napoca system-wide protection activation and protected process registration can be done using a hypervisor interface provided by BitDefender to dynamically control the hypervisor and the introspection library behavior. The interface can be accessed from the command line through a user-space hypervisor controller provided together with Napoca by BitDefender. Cross-domain communication between ring 3 and ring -1 is implemented leveraging an extra layer represented by a kernel module installed in the guest VM at Napoca configuration time. The userspace

application invokes the kernel through IO facilities to ask the driver to relay messages to the hypervisor. Its commands are described at [15]. The first command invoked to set up monitoring is the *updateflags* command which installs protection options using a numerical flag. Such a value is a bitmask composed `XOR-ing` numerical values that represent a different protection option each as described in [15]. Listing 6.2.2 depicts the userspace protection activation command.

```
1    updateflags 0x18300004000
```

Listing 6.9: Prompt to activate while userspace protection on HVMI.

The numerical constant following the command in Listing 6.2.2 corresponds to the `PROC_OPT_PROT_ALL` flag, which sets all the protection options for the user-mode, XOR-ed with a flag to activate an optimization which will be described later. The use of the above flag is mandatory to enable code unpacking protection for designated processes. Subsequently, we use the *protect* command which takes as input the name of a process, a protection mask, and a process identifier for the logs. It activates the chosen protection measures for the process.

```
1                protect process.exe 0x8 1
```

Listing 6.10: Protect process.exe against unpacking in its code section on HVMI.

In the example of Listing 6.2.2, the used protection mask is 0x8, which is the value of the `PROC_OPT_PROT_UNPACK` flag in HVMI code [15]. The *protect* command triggers the execution of the *IntAddRemoveProtectedProcessUtf16* HVMI API. This function stops guest execution and checks on the family of operating systems HVMI is monitoring. Upon OS identification, they call the Introcore API specific to the monitored operating system in charge of enabling process protection [18]. What HVMI does with processes is a form of context-sensitive monitoring strictly integrated into the operating system to minimize the overhead of the introspection. To avoid CR3 hooking, which causes a VM-Exit at each context switch, HVMI resorts to massive use of detours and write hooks. If the process to be monitored is active, the introspection library iterates through its internal representation of the active processes and updates it accordingly to the new protection flags. Actual protection is activated by the principal API for process protection, *IntWinProcChangeProtectionFlags* When the process to be monitored has not yet been started, the library has to resort to a quite different strategy. At userspace protection activation, a detour on the Windows internal routine *PspInsertProcess* is installed. Such a function is responsible for inserting a newly created process inside the list of active processes in the Windows kernel. The detour callback sets up the necessary data structures for starting

the monitoring of the memory of the process, but the hooks are only installed when the just-created process is swapped in by the operating system.

The `EPROCESS` structure of a Windows process is the kernel data structure collecting all the information needed for the process to be scheduled and run. HVMI hooks write accesses to one flag named `EPROCESS.OutSwapped` to be informed when the process is ready to be loaded in memory. When this happens, HVMI places a detour on *MmInSwapProcess* Windows kernel function to trace process loading. Its handler creates an internal view of the process to be run and invokes the function *IntWinProcChangeProtectionFlags* above mentioned. Figure 6.6 explains the protection activation flow taking as an example a fictitious process called A. Dashed lines represent asynchronous events.

To detect code unpacking inside the memory of the process to protect, HVMI invokes *IntWinModHookPoly* on the Introcore internal representation of the process protected with the `PROC_OPT_PROT_UNPACK` flag [17].

```
static INTSTATUS IntWinModHookPoly( _In_ PWIN_PROCESS_MODULE Module )
```

Listing 6.11: Signature of IntWinModHookPoly.

This latter routine identifies the sections of the memory areas of the process that may contain code, finds the pages on which such sections are mapped, and invokes *IntUnpWatchPage* on every one of them. The data relative to the executable of the process is stored in the `PWIN_PROCESS_MODULE` structure given in input to *IntWinModHookPoly*.

By default, *IntUnpWatchPage* installs a write hook on the page of interest with the command in Listing 6.12.

```
status = IntHookGvaSetHook(Cr3,
                           VirtualAddress,
                           PAGE_SIZE,
                           IG_EPT_HOOK_WRITE,
                           IntUnpPageWriteCallback,
                           pPage,
                           NULL,
                           0,
                           (PHOOK_GVA *)&pPage->WriteHook);
```

Listing 6.12: Installing a memory hook on a guest virtual address in HVMI.

The registered callback *IntUnpPageWriteCallback* is triggered upon write access to the page and updates its internal state according to the logic depicted in Figure 6.7. Dashed lines indicate that the transition may just never happen.

HVMI checks whether the offending write is valid or not. Write validity depends on the location where it happened, because some locations of the monitored executable may be written by system routine to locate libraries loaded and called by the program, but such memory locations are not executable. A write is invalid when it concerns executable memory. In the case the write is not valid, the page is internally marked as "dirty" in the write hook callback and a counter is incremented. When the value of this counter becomes greater than a threshold (default:32), the write hook is swapped with an execute one whose callback is *IntUnpPageExecuteCallback*. On the first instruction fetch from the page, the execute hook fires, and HVMI raises an alert to the user. In such a case, the process is forcefully terminated [3] [3].

To carry out our test and implement the same logic as Drakvuf inside HVMI, we applied the following modifications in order:

- We set the threshold for the write hook to 1
- Every write is considered invalid always forcing the write validity flag to false
- *IntUnpPageExecuteCallback* no longer forcefully terminates the program. Instead, it marks the page as clean and reinstalls the hook on write accesses, associating with them the *IntUnpPageWriteCallback* callback.
- A counter has been created to track every invocation of the execute access callback. This has helped during the development of the patch to check that the hook was activated as many times as the iterations of the $\mathbb{W} + \mathbb{X}$ loop in the micro-benchmark.

6.2.3. Monitoring with HyperDbg

The HyperDbg test has been carried out debugging manually the monitored program. The program is started using the *.start* debugger command followed by the path of the executable inside the guest. Leveraging introspection APIs, processes can be created by code injection in the guest and the debugger can automatically attach to them single-stepping the operating system routine used to create the process until the entry point of the binary is known. At that point, by default, HyperDbg installs a breakpoint at the program entry and stops the execution of the entire guest as soon as the execution flow reaches the entry point since a VM-exit brings control to the hypervisor. Upon stopping, another breakpoint is installed on the CALL instruction which is used to invoke the *VirtualProtect* function. After the breakpoint fires, we read the register RCX to get the virtual address of the just initialized page whose permissions are going to be

[3] HVMI source code: https://github.com/bitdefender/hvmi

changed. Subsequently, we step past the instruction. While carrying out experiments, having reached this point of the program execution, we observed that the newly allocated page has been loaded in memory and this is confirmed by the fact that an EPT-based hook has been successfully installed on the entire page. The fact that the page has to be effectively allocated in guest physical memory is pivotal to placing a hook on it, since, different from HVMI which automatically handles lazy loading and swapping, HyperDbg cannot install hooks on the page tables to trace page mapping to guest memory frames following a page fault. After the installation of the hook, the program is resumed. The installed hook removes both write and execute access from the monitored page. It is placed in the guest using the *!monitor* command in the HyperDbg console. Such a command can be used to monitor a contiguous range of memory addresses in the guest, with no size limitation, assuming that all the virtual addresses included in the specified range are valid to the guest. By design, HyperDbg implements the *!monitor* feature directly manipulating the permissions of the hooked EPT pages to remove the one of interest to the monitoring. Upon illegitimate access to a hooked page, an EPT violation occurs and the guest traps in the hypervisor to inform it that the page was accessed. Control is given back to the guest after setting the Monitor Trap Flag and restoring the original permissions of the page. The offending instruction is re-executed on the page with restored permissions and, right after the execution, the MTF triggers a new trap in the hypervisor. This second exit is managed by disabling MTF on the monitored core and removing the permissions from the page again to reactivate the hook [8]. This means that, for each violation, two exits to the hypervisor have to be performed.

```
1    !monitor rwx $start $end stage pre buffer 0x40 script{
2        // script body
3    }
```

Listing 6.13: Invocation of the !monitor extended command in HyperDbg.

The command in Listing 6.2.3is followed by two addresses describing the starting and the ending point of the memory range to be monitored, since it is a hypervisor-level tool for debug register emulation as explained in the 2. The "stage pre" directive specifies that the trap to the hypervisor has to happen before the execution of the offending instruction, i.e., the instruction that caused the exit to the VMM. Following the stage, another directive asks the hypervisor to reserve a pre-allocated buffer of size 64 bytes (0x40 in hex) for the event just registered. Finally, the script specifies the callback that is associated with the exit to the hypervisor in response to the violation.

Script 6.14 implements the logic described in chapter 5. The $context variable keeps a reference to the guest virtual address at which the violation happened. If it is different

from the instruction pointer of the guest (RIP) a write has been detected and the page is marked as dirty writing the quadword 0x1 inside the first 8 bytes of the preallocated buffer using the *eq*. This quadword describes the state of a memory page at any given point in time. At the next trap, if the $context variable has the same value as RIP, the value of such buffer location is checked using the *dq* command to read the quadword. If the page is dirty (quadword set to 0x1) an execution has been detected and the page is marked as clean to restart the test. Debug prints signal the branches associated with each event but for the test, they have been commented out since I/O operations add overhead to the monitoring and slow down the debugger.

```
!monitor rwx $arg1 $arg1+fff stage pre buffer 0x40 script{
    if($context == @rip ){
      if(dq($buffer) == 1){
          //printf("EXECUTE access at %llx\n", $context);
          eq($buffer, 0);
      }
    }
    else{
        //printf("WRITE access at %llx\n", $context);
        eq($buffer, 1); // mark as dirty
    }
}
```

Listing 6.14: xwt.ds : implementing monitoring behavior on HyperDbg

6.3. Macro-benchmark

The macro-benchmark has been implemented as a Python script which uses the **pywinauto** library to invoke Windows UI automation APIs. Windows UI Automation (UIA) is an automation framework provided by Microsoft that allows developers to access and manipulate the user interface elements of Windows applications programmatically. UIA provides a set of APIs that enable interaction with UI controls, retrieving information about them, and performing actions such as clicking buttons or entering text. The **pywinauto** library is a Python wrapper around the UIA APIs, providing a convenient and easy-to-use interface for automating Windows applications [39] [47]. The pywinauto library enables process creation and interaction with existing processes in Python scripts. Each UI interaction revolves around a data structure that describes the state of the UI for the current user session providing a hierarchical view of the GUI elements, i.e., the tree of the graphical elements. This data structure can be inspected using a Python script provided by the authors of the tool. An example screen capture of such a GUI inspection

```
def test_np():
    notepad = Application().start("notepad.exe")
    notepad.UntitledNotepad.wait("ready")
    notepad.UntitledNotepad.Edit.type_keys("Test")
    notepad.UntitledNotepad.menu_select("File->Save As")
    notepad.SaveAs.edit1.wait("ready", timeout=20) // fail-safe time
   margin
    notepad.SaveAs.edit1.set_text("example.txt")
    notepad.SaveAs.Save.click()
    if notepad.ConfirmSaveAs.exists():
        notepad.ConfirmSaveAs.Yes.click()
    notepad.ExampleNotepad.menu_select("File->Exit")
    notepad.ExampleNotepad.wait_not('visible', timeout=10)
    return
```

Listing 6.15: Example function written to automate Notepad testing

tool is reported in Figure 6.8.

Such a memory view can be used as a reference when implementing GUI interaction to learn the name of the graphic components and their hierarchical organization on the graphical interface. The information extracted from py_inspect.py is directly used to write interaction scripts. In the scripts, the pywinauto library converts Python object attribute references to paths used for GUI tree traversal in the UIA tree-like elements hierarchy. For example, *notepad.UntitledNotepad.Edit.type_keys("Hello World")* is a valid Python command that iteratively looks for an object with a name similar to "notepad" in the first level of the graphic elements tree. After the first match, it searches for notepad window (second level of the tree) components whose name can match "UntitledNotepad" and, after finding such a component, it looks for an "Edit" component inside the second level one. If one is found, the *type_keys* method is called on it to type the string "Hello World" in the context of the "Edit" component, which may, for example, be a text box. Name matching is carried out using an ad hoc approximate string matching algorithm [47]. The most important part of the macro-benchmark per-application routines written is the insertion of sleep and waits. As stated in chapter 5, monitoring actions introduce a slowdown in the rendering and display of Windows elements. The script for automatic interaction has to be aware of such delays so that it can wait for the graphical elements to be available to be interacted with. As an example, the interaction script 6.15 has been used to test the Notepad application.

As reported, after starting the application, the script waits for the GUI to be visible in the foreground. The maximum timeout value, i.e., the maximum wait, is fixed to 20 seconds as an extra safe margin. When the app window is visible, the interaction starts. Similarly, when new dialogs and windows are opened, the other scripts wait for them to be visible

before interacting with them.

Every per-application GUI simulation routine is launched by the *benchmark_perfcounter* routine after taking a time sample from performance counters, special CPU registers to be used for fine-grained time measurements [5]. At the end, another time measurement is taken and the difference between the two measurements is computed to get the duration of the routine. The duration is then printed to the standard output. Time is measured using performance counters since they are the most precise and reliable way of taking time measurements in Python [10].

6.3.1. Monitoring with Drakvuf

Macro-benchmarking in Drakvuf has been implemented using the Codemon plugin, a Drakvuf extension designed to automate hidden code unpacking detection and its extraction using the Xen hypervisor. The plugin is started and configured at the beginning of a Drakvuf session specifying the name of the plugin and some parameters to establish the default behavior in case of potential unpacking attempts, the filesystem path where to store analysis logs and dumped memory pages, and the verbosity level of the logs. Dumps can optionally be disabled by setting the "benign" behavior in Codemon. Such behavior instructs Codemon monitor to consider benign every potential unpacking attempt, detected using the approach described in chapter 3. Since they require intensive I/O operations to read guest memory and write it on the disk, making them unsuitable for real-time monitoring, we set benign behavior at every test hereby described when starting Drakvuf with the Codemon plugin on. Moreover, to avoid noise in the measurements and focus on assessing the performance overhead on the single process, we set the context-sensitive monitoring mode on the process to be tested. Listing 6.16 depicts the prompt used to start Drakvuf monitoring:

```
1  sudo ./drakvuf -d win10 -r win10.json -a codemon \
2                                        --codemon-dump-dir /tmp \
3                                        --codemon-default-benign
4                                        -C --context-process process.exe
```

Listing 6.16: Invocation of Drakvuf with the Codemon plugin.

During its initialization, the Codemon plugin puts a hook on the MmAccessFault routine. This routine handles page faults in the Windows kernel, so, it is called upon access to pages not yet loaded due to the lazy memory loading strategy in modern operating systems. Along with this routine, the constructor also hooks KiSystemServiceHandler, the kernel exception manager. Sometimes, while the tool is used, it may erroneously

trigger page faults in a non-paged area of the kernel. This results in the so-called "blue screen of death" in Windows. To prevent that from happening, this routine is hooked and whenever it is called, its execution is ignored using VMI provided that Codemon can make sure that the cause of the invocation is linked to Drakvuf monitoring. Focusing on the page fault handling routine, the callback associated with the MmAccessFault trap is called *mm_access_fault_hook_cb*. This function has the primary goal of installing a return hook on the MmAccessFault routine invocation to track the guest physical page where the accesses not-yet-allocated page is allocated by the OS. The return hook is placed by Drakvuf simply reading the value of the RSP register, which points to the top of the stack in x86 processors. That location hosts the next instruction pointer value of the caller routine. When the hook on the saved RIP fires, the allocated page physical page address returned by MmAccessFault can be extracted and used to install a hook on the page.

The routine *mm_access_fault_return_hook_cb* is in charge of installing such an execute hook with the command in Listing 6.17.

```
1  auto exec_hook = createMemAccessHook<AccessFaultResult>
2            (&codemon::execute_faulted_cb, gfn, PRE, VMI_MEMACCESS_X);
```

Listing 6.17: Creation of a memory hook.

One important catch is that, as suggested for HyperDbg, the page must be mapped to a guest frame in memory. As this may not always be the case following a MmAccessFault invocation, Codemon injects a page fault in the guest to force page loading and restarts guest execution. This is made possible by the capabilities of the Xen hypervisor as described in chapter 2. As soon as the guest operating system invokes the MmAccessFault routine to bring the page to the guest physical memory following the injected page fault, the previously installed return callback checks whether it has been mapped to a frame or not. If this did not happen, no further page faults are injected and the page is ignored since Codemon was not able to retrieve it for unknown reasons. Otherwise, the write hook is installed as depicted above. MemAccessHook is a page-wide hook implemented through a modification of the permissions of the memory page of interest in the `idx` view, without duplicating memory and resorting to different views as happens for stealthy breakpoints. The callback associated with the execute hook is *execute_faulted_cb*.

Upon the first execute access to the page, in our testing flow, what the routine does is place an identifier of the page and the relative CR3 register value inside the set of monitored pages. This happens because the tool infers that such a page is going to contain code from the fact that instructions are fetched from it. Subsequently, such callback swaps

```
mmvad_info_t mmvad;
if (!drakvuf_find_mmvad(drakvuf,
                        trap_info->proc_data.base_addr,
                        fault_va, &mmvad))
{
    PRINT_DEBUG("[CODEMON] Could not find vad information\n");
    return false;
}

auto vadname = drakvuf_read_unicode_va(drakvuf,
                                mmvad.file_name_ptr, 0);

if(vadname!=NULL &&
                vadname->contents!=NULL &&
                strstr((char*) vadname->contents, "dll" )){
    PRINT_DEBUG("No hooks on \acrshort{dll}s!\n");
    return false;
}
```

Listing 6.18: Patch to the Codemon plugin to rule out DLL hooking.

the execute hook with a write one to be handled by the *write_faulted_cb* callback. To identify monitored pages and keep track of them, the *execute_faulted_cb* calls a routine named *analyze_memory*. It computes the actual page identifier, a SHA256 hash digest. Moreover, the routine defines a hook for the insertion of a malware detection tool to monitor the page content. These elements are not of interest to the current thesis, so they will not be further examined.

To rule out DLL hooking for the reasons described in chapter 5, MmAccessFault has been patched including the following snippet: The name of the executable whose page has been requested is retrieved using the function *drakvuf_find_mmvad*. It retrieves the in-guest data structure MMVAD including the data about the virtual address space where the faulting page belongs. Such data includes the name of the executable loaded at that address space. This is extracted by reading a string in the guest using the *drakvuf_read_unicode_va* function. It may be that the library cannot retrieve such datum, so we check if the retrieval went well, and, in such case, we check for the presence of the string "dll" inside the name. If it is found, the page is ignored. Listing 6.18 includes the patch.

6.3.2. Monitoring with HVMI

Implementing the macro-benchmark on Napoca and HVMI involved using the same process protection installation logic as the micro-benchmark as explained in section 6.2.2. Listing 6.19 describes the inserted prompt to monitor the Calculator application:

```
protect calc.exe 0x8 2
```

Listing 6.19: Setting a protected process on HVMI.

At the startup of one of the protected processes, HVMI hooks its memory and the monitoring effectively starts. Process identification and hooking are realized in the same way as the micro-benchmark. The same script used while doing the macro-benchmark on Windows has been employed for testing processes. It is important to stress that Introcore only places hooks on the process regions that are the memory mapping of the various sections of the executable in memory, without dynamically hooking shared code regions whose location is resolved at runtime. The unpacker module used to implement unpacking protection at page granularity does not dynamically trace memory accesses to memory mappings subsequently added to the virtual address space of the process. Consequences are two-folded:

- As proved with experiments, when dynamically allocating memory which is both writable and executable, unpacking is not detected, i.e., the monitoring can be easily evaded. Experiment results proving this fact are reported in chapter 7.
- Access to the pages of DLLs cannot be tracked. This is consistent with the way we implemented memory tracing on Codemon.

As of our understanding, there is no ready-to-use way of tracking dynamically allocated memory for code unpacking in HVMI. Even though we believe that there exist many ways of implementing such a feature in HVMI, it is out of the scope of the present work.

6.4. Time collection

Time measurements in the micro-benchmark have been taken using the C++ high-resolution clock. The clock is the most precise and reliable way of taking time measurements in C++ [1]. It ensured interoperability and compatibility among different Windows versions since the same language runtime has been used thanks to the static-linking of the micro-benchmark program. In the case of the macro-benchmark, time measurements have been collected using performance counters. They are the most precise and reliable way of taking time measurements in Python [10] and, since the same Windows version and Python runtime have been employed on both Napoca and Xen for the tests, we assume that the used methodology is reliable and consistent. Timing data coming from different time sources are not compared with each other.

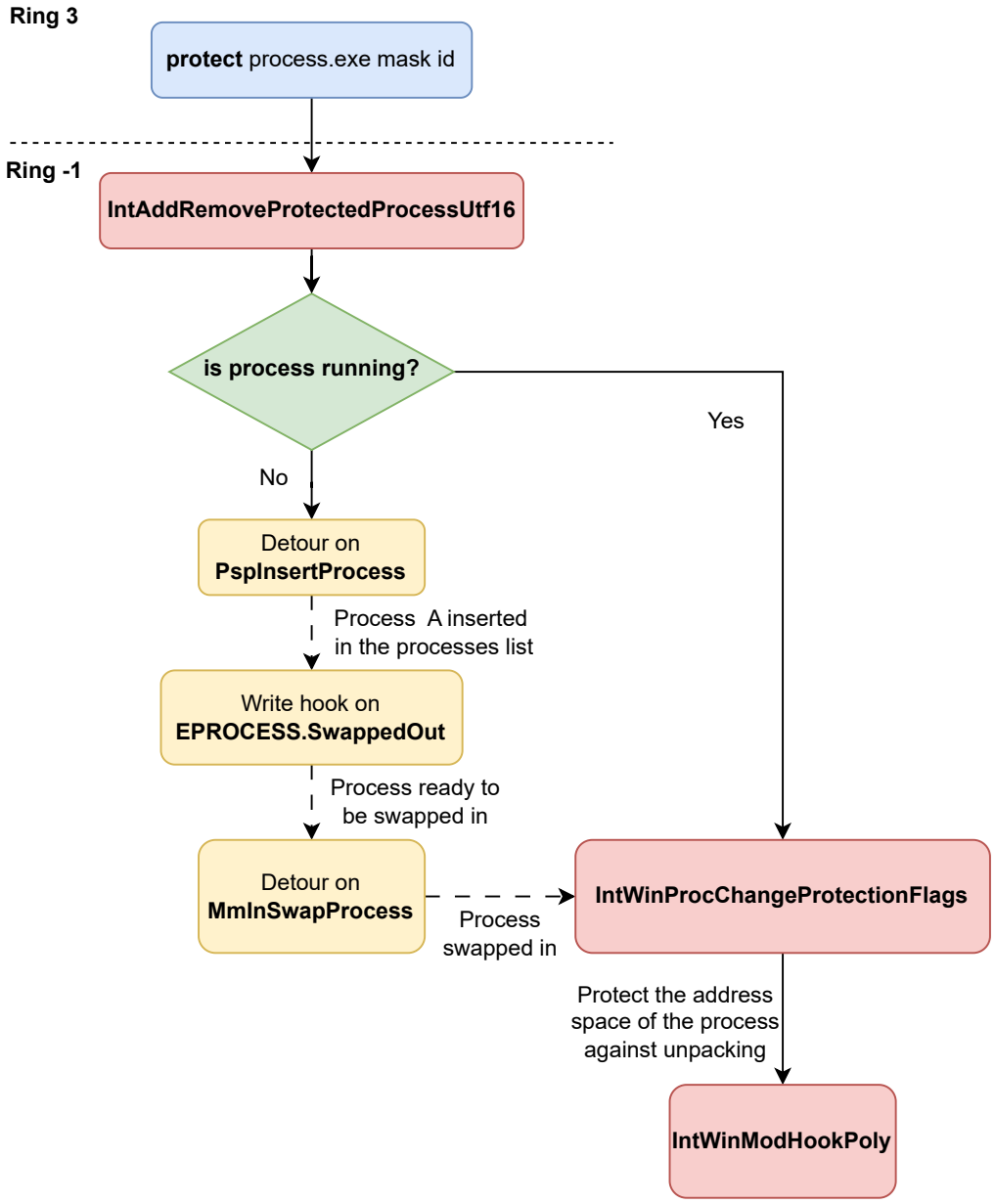

Figure 6.6: Installation of A as a protected process in HVMI.

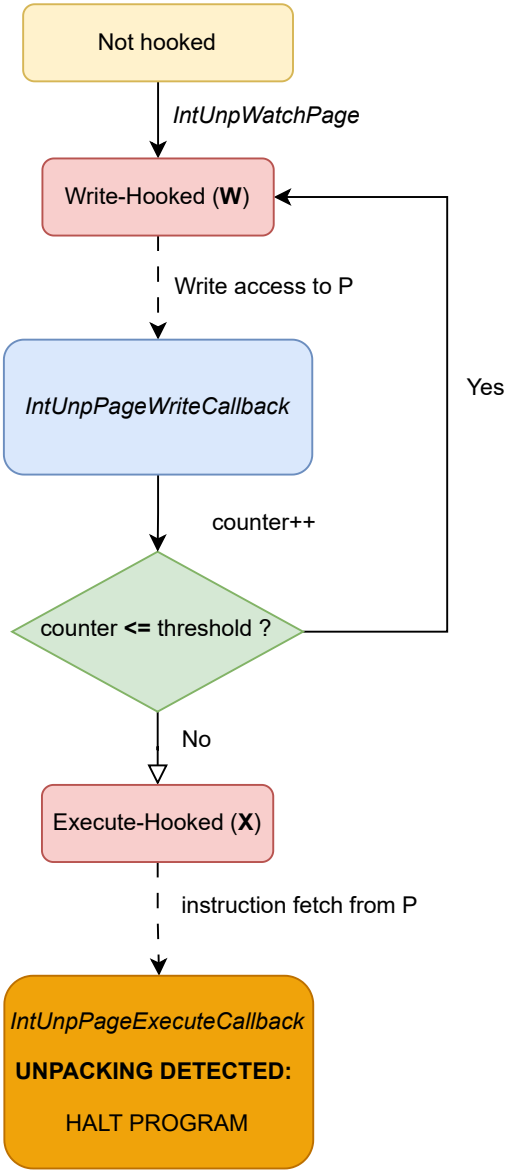

Figure 6.7: Evolution of monitoring state on page P allocated to a protected process.

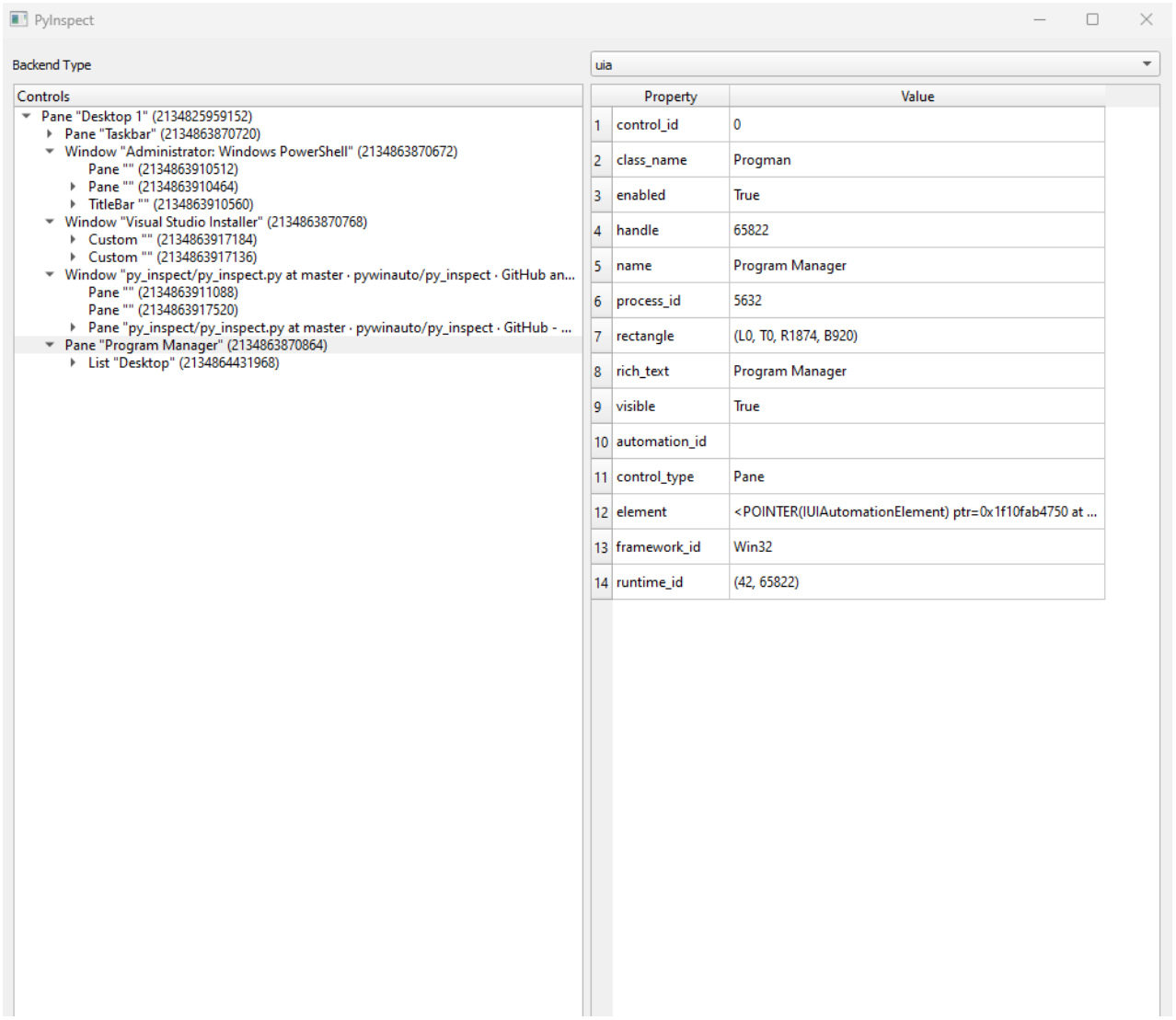

Figure 6.8: Screen capture of the Windows GUI inspection tool py_inspect.py

7 | Experiments

This chapter explains and presents the experimental part of the present thesis work. To answer our research questions, we first implemented several experiments aimed at providing a characterization of the slowdown incurred by memory accesses in the presence of EPT-based memory hooks and trap swapping for guest execution tracking.

We compare the data to identify the best-performing approach. In light of this comparison, we test Drakvuf and HVMI on real-world applications to understand what is the performance of these latter and which approach has the lowest overhead on the monitored system. We wish to understand whether the slowdown of memory operations reflects or is, at least, numerically close to the one encountered during real-world application testing. Apart from that, we aim to investigate one of the blind spots of HVMI and discuss its impact on the precision and reliability of an unpacking detector at the hypervisor.

At last, we try to retrofit the HyperDbg debugger as a memory monitor for code unpacking detection. The experiments to be presented aim to investigate the proposed strategy for monitoring with HyperDbg from a functional and performance standpoint, comparing it to the other tested approaches. Finally, we analyze the scalability of HyperDbg-based monitoring, pointing out its limitations.

The present chapter also includes a set of experiments on the evaluation of the time taken by the Xen hypervisor to perform guest page table switching. We have pointed out in previous chapters that such a strategy helps avoid the burden of implementing emulation and complex strategies for handling traps in the hypervisor in the context of hypervisor-based monitoring. Here we wish to point out what is the cost of alternating page tables on the simplest virtual machine possible from an emulated hardware configuration standpoint. This experiment aims to shed some light on the intrinsic cost in terms of time for implementing page table alternation in monitoring solutions.

Testing environments

Table 7.1 presents the setups associated with each of the experiments here described.

	altp2m	**Drakvuf**	**HVMI**	**HyperDbg**
Guest Operating System	Ubuntu 20.04 Linux 5.15.0.92	Windows 10 build 17763	Windows 10 build 17763	Windows 11 build 22631
Hypervisor	Xen 4.17.2 (patched) branch : master commit: 322a20add0	Xen 4.17.2 branch : master commit: 322a20add0	Napoca branch : master commit : 9f266d3	**HyperDbg 0.8.1 branch : master commit : 68e0d32
Host Operating System	Ubuntu 20.04 Linux 5.15.0.92	Ubuntu 20.04 Linux 5.15.0.92	Windows 10 build 17763	Windows 10 build 17763
Python version		3.12.1	3.12.1	
CPU cores allocated to guest	1	2	4	2
Guest RAM	3500 MB	3500 MB	8192 MB	4096 MB

Table 7.1: Hardware and software configuration of the employed virtual machines.

We list some considerations and details on the testing environment:

- The same physical machine was used to do all the tests.
- HyperDbg does not work properly on the same Windows version as the one where the other solutions were tested. Since it has been extensively tested on Windows 11 23H2, we decided to move to that Windows version.
- A different operating system has been used to carry out the altp2m test because we wanted to use a single-core virtual machine and none of the tested Windows versions booted in Xen with only a single vCPU allocated to the machine.
- The micro-benchmark executable has been compiled and statically linked using Microsoft Visual Studio 22 and the Microsoft Visual C++ Redistributable language runtime, version 14.38.33135.0. Linking the C++ runtime statically inside the executables allowed us to prevent runtime compatibility and portability issues potentially associated with platform variability.
- The same 11th Gen Intel Core i5-1135G7 2.40GHz CPU was used in all the setups. It is equipped with HyperThreading technology, which allows the execution of two hardware threads per physical core [5].
- The available host RAM was 8GB. Guest RAM values in the case of Xen have been fixed to that threshold to limit performance degradation due to excessive swapping. Since physical RAM was limited to 8GB, and normal Drakvuf and Xen Host operation almost filled 4GB during the tests and excessive swapping may hurt performance, we empirically found this threshold to reach a compromise between performance and memory usage. In the case of HyperDbg(**), the VM is running on

VMware Workstation Hypervisor version 17.0 before the installation of the HyperDbg hypervisor. Also in this case, the reason why the RAM has been capped at 4 gigabytes is that the remaining 4 gigabytes are needed by the host operating system. This threshold has been found empirically.

- The Python version field has not been filled for HyperDbg since no Python script was used in the tests.

7.1. Evaluating the cost of altp2m

As explained in chapter 6 and claimed by Lengyel at al. [34], altp2m is a game-changing mechanism for hypervisor-based stealthy monitoring as it may **ease** the implementation of efficient, stealthy hypervisor-based monitoring. It is at the base of the tool Drakvuf, analyzed in the present thesis work. The goal of this set of experiments is to estimate quantitatively what the overhead associated with **page table switch at the hypervisor level** is. The userland part of the experiment logic depicted in chapter 6 has been implemented in a program named "altp2m-benchmark.c", whose code has been reported in Appendix A.

The average time taken by the altp2m switch to execute amounts to 5040.53 nanoseconds. It is the mathematical average of 500000 execution time samples. They have been collected using a Bash script which calls the benchmark program 250000 times because one run of the benchmark triggers the activation of the routine *p2m_switch_domain_altp2m_by_id* twice. We decided to implement this test using a single-vCPU virtual machine to put ourselves in the simplest context possible as we believe that by using a single-vCPU setup, we gain insights into the absolute minimum price to pay for page table alternation on a running virtual machine. Results collection requested periodical flushing of the Xen log buffer because prints filled it before the end of the benchmark and data written post overflow of the buffer overwrites its previous content. To interpret the altp2m datum about the time taken by Xen altp2m page table switching, we can compare it to the time taken to execute one iteration of the $\mathbb{W} + \mathbb{X}$ loop described in chapter 6 computing a ratio between them, whose value is 44.74. It is the result of 5040.53 divided by 112.70, this latter being the time for one iteration of the $\mathbb{W} + \mathbb{X}$ loop on Xen without monitoring.

The methodology used to estimate the time taken by one iteration of the $\mathbb{W} + \mathbb{X}$ loop is explained in section 7.2. Factor 44.74 represents how the time cost of alt2pm compares to the execution of a write and an instruction fetch in a guest virtual machine.

Even though, as already explained in chapter 6, such time measurements are taken from

different runtime environments and may not be directly comparable, we believe the estimated factor helps us to understand that dynamically switching page tables may result in an overhead that is not negligible. More specifically, the presented datum provides a quantitative measure that helps understand in advance that, depending on the frequency of the traps that are managed by switching page tables, the performance of the guest operating system may be considerably reduced. We believe this is a starting point to begin building a more complete picture of the "price" of altp2m-based hypervisor-based monitoring. The slowdown factor extracted from the experiment on the characterization of altp2m is **44.74x**. This means that switching the page table pointer on a single-vCPU virtual machine on the Xen hypervisor is **44.74** times slower than two instruction fetches (a MOV and a RET) on a virtual machine. The second element was considered for the comparison for convenience, as the data about it was already available after the data collection for the micro-benchmark experiment on Xen. We believe comparing the altp2m datum with the time measurement of such a simple operation sequence helps us understand the impact on guest execution of inserting a page table switch synchronous to guest operations. This being said, as measured and proved by Suneja et al. [50], the addition of VMI operations to solve the semantic gap problem incurs a non-negligible overhead that also depends on the approach on which they are based. So, even if the data about altp2m and how it compares to a simple operation sequence may help to shed some light on hypervisor-based monitoring solutions using such a feature, further work is needed to provide a complete performance characterization of hypervisor-based monitoring solutions for code unpacking detection.

7.2. Micro-benchmark

In this section, we want to quantitatively **characterize the slowdown of memory accesses** in the presence of EPT-based introspection-powered memory hooks. We are willing to **assess the performance** of the two approaches to solve the problem of interest presented in chapter 5 and reported in Listing A.2 in Appendix A to shed some light on how the **hooking and trap switching strategies** of such solutions impact the overhead on the monitored virtual machine.

- Drakvuf: altp2m-based memory view alternation with hypervisor-level permissions alteration
- HVMI: EPT pages permissions alterations with one single memory view

To this aim, the micro-benchmark program presented in chapter 6 has been tested on Xen and Napoca both with monitoring off and on. The name of the executable implementing

the base version of the micro-benchmark is **unpacker.exe**. The micro-benchmark presented in chapter 6 is first run on the Xen virtual machine described in Table 7.1 without monitoring. It is started from the Powershell command line using the prompt in the Listing 7.1.

```
.\unpacker.exe 500000
```

Listing 7.1: Prompt to invoke the micro-benchmark.

After a run, the results are output on the screen and collected. The process is repeated enabling Drakvuf-based monitoring with the bash prompt 7.2.

```
./drakvuf  -d win10 -r /root/win10.json -a codemon
        --codemon-default-benign
        --codemon-dump-dir /tmp
        -C --context-process $proc.exe
        -i $pid_for_injection
        -e "python.exe C:\\test.py $proc $n"
```

Listing 7.2: Drakvuf invocation prompt

The flags in the prompt starting with the prefix "codemon" configure the plugin to create a directory for potential memory dumps under "/tmp", even if dumping is disabled with the "default-benign" flag. The "-C" flag is used to enable the context-sensitive monitoring mode and "–context-process" signals the name of the process to monitor that it is the value of the $process variable. In prompt 7.2, the placeholder $proc is the name of the process to be monitored, $n is the number of iterations of the $\mathbb{W} + \mathbb{X}$ loop and $pid_for_injection is the process ID of the process that is hijacked to launch the program Python to execute the script test.py, as the argument reported after the "-e" flag suggests. The testing methodology here employed leverages Drakvuf shellcode injection capabilities for injecting a process inside a virtual machine hijacking a running process starting from its PID documented in [34]. This methodology has been used for ease of testing automation.

	Loop execution time	**Mean per-iteration time**
Xen Monitoring OFF	56348600 ns	112.70 ns
Xen Monitoring ON	119488721739 ns	238977.44 ns
Slowdown Factor	2120.527x	

Table 7.2: Results of monitoring on Xen.

Upon collection, the time taken by the micro-benchmark to execute while monitored is compared to the time taken by its run without monitoring, and the slowdown factor is

computed as the mean per-iteration time in the case of monitoring ON and the analogous value in the case of no monitoring.

The same experiment is repeated on Napoca and HVMI and the results are compared to Xen and Drakvuf to investigate the differences between the two approaches. The micro-benchmark is first run on Windows running on Napoca without monitoring solutions activated. Then HVMI-based userspace protection is installed and the **unpacker.exe** program is protected against unpacking.

```
protect unpacker.exe 0x8 1234
```

Listing 7.3: Prompt to install unpacking protection for the micro-benchmark.

The results of monitoring on Napoca are presented in Table 7.3.

	Loop execution time	**Mean per-iteration time**
Napoca Monitoring OFF	64294844 ns	128.59 ns
Napoca Monitoring ON	142273657753 ns	284547.32 ns
Slowdown Factor	2212.527x	

Table 7.3: Results of monitoring on Napoca.

Analogously to what has been done with Xen, the slowdown factor is computed as the mean per-iteration time in the case of monitoring ON divided by the analogous value in the case of no monitoring.

It is important to stress that the whole user-space monitoring has been disabled before executing the micro-benchmark in the "Monitoring OFF" case, i.e. we did not just disable protection for the single process. This has been done for consistency with the Xen case study, where when Drakvuf is off, no external introspection agent is interfering with any event inside the machine.

For Napoca and HVMI, **two** alternative configurations for userspace monitoring have been tested. In the first configuration, user mode protection is activated factoring out the optimization mentioned in chapter 5, while in the second case, the optimization is included. Such optimization has already been described in chapter 2 and it is aimed at reducing the number of exits to the hypervisor while monitoring with HVMI on Napoca. HVMI installs hooks directly on the page tables and such hooks may fire when the OS pager updates metadata and not the data needed to describe page translation of the virtual addresses of the guest. If an entry is write-hooked and the OS pager accesses that entry to modify the value of a flag, without changing the actual value of the page table entry, the hook will

fire as happens with actual modifications to page physical mapping data. Nonetheless, this is considered a spurious VM-Exit because the intervention of the hypervisor is not required. To limit spurious VM-Exits, upon configuration, Napoca loads in the guest hijacking a privileged process a special kernel module which is in charge of filtering out traps to the hypervisor that are not relative to permissions violations on hooked memory regions. This trap filtering logic is implemented leveraging Intel Virtualization Exception to translate traps in the hypervisor to interrupts raised in the guest operating system. The handler for such interrupts is the Virtualization Exception (#VE) Agent installed by HVMI as a kernel module which verifies the cause of the exit and issues a hypercall if the exit is linked to a hook being activated. Comparison between the two prompts is reported in Listing 7.4.

```
1  updateflags 0x183c80030000 (1)
2  updateflags 0x183c00030000 (2)
```

Listing 7.4: Different "updateflags" invocations with #VE optimization (1) and without (2).

The only difference is that the first flags combination has been XOR-ed with the value 0x8000000000 which, parsed by the hypervisor, triggers #VE agent injection. There being two alternatives, we asked ourselves how much the Virtualization Exception optimization influences the performance overhead on the single memory operation associated with hypervisor-based monitoring on Napoca. To answer this question we run the micro-benchmark in both the case with #VE agent and the case without it. The results are summarized in Table 7.4:

	Loop execution time	**Mean per-iteration time**
#VE agent ON	142273657753 ns	284547.32 ns
#VE agent OFF	142941044118 ns	285882.09 ns

Table 7.4: Results of monitoring on Napoca with and without #VE agent.

The delta between the two per iteration mean times is **1334.77 ns**. According to the data, the optimization improves the mean per-iteration time. The improvement can be expressed in percentage as the ratio between the delta and the mean per-iteration time in the case of monitoring without the #VE agent as follows:

The ratio between the delta and the maximum mean per-iteration time is `1334.77 /`

`285882.09 * 100 = 0.467`, suggesting that the improvement is negligible. This suggests that the use of
Virtualization Exception-based optimizations to account for excessive vm-exits does not impact the slowdown of the guest memory operations stemming from such exits. If not designed properly, the injection of an agent inside the guest may reveal to the malware the presence of a hypervisor-based monitoring solution. This may represent a potential menace to system security, given the behavior of malware reported in chapter 2. On this basis, we can conclude that, given such a relatively low speed-up, the use of Virtualization Exception for aiding system monitoring is not worth the risk. Nonetheless, we opted for using the #VE agent optimization in all the tests carried out on Napoca.

Given the slowdown factors of Drakvuf and HVMI, we can conclude that trap switching for fine-grained memory page access tracing incurs a slightly lower performance overhead in Drakvuf than in HVMI.

The ratio between the slowdown factors (**2120** for Drakvuf and **2212** for HVMI) of the two monitoring solutions is **1.043** meaning that HVMI monitoring results in a 4.3% higher performance overhead than Drakvuf.

The most prominent difference between HVMI and Drakvuf is that trap handling and callback execution happen at the hypervisor level in the first case and in user space in the second case. Managing trap in the host requires an extra communication step to pass trap data to the user space library, different from what happens in HVMI, which runs at the hypervisor level and directly invokes it in response to violations. Nonetheless, this difference does not seem to determine a consistent performance improvement with respect to Drakvuf in the case of HVMI. On the contrary, this datum suggests that altp2m-based page table switching is a slightly more efficient design choice for implementing EPT-based memory hooks for memory monitoring than simple page table permissions alteration and emulation in response to violations. However, since the difference is not very pronounced, we cannot draw strong conclusions on which approach is the absolute best for EPT-based hooks and trap-switching implementation.

7.3. Macro-benchmark

The efficiency of tools for memory access tracing for code unpacking detection is influenced by the approach to memory hooking and by the strategy used to solve the semantic **semantic gap**. This section investigates the performance of the solutions presented to solve this latter and to track closely the execution of programs inside the virtual machine.

The experiments to be presented are based on the macro-benchmark described in chapter 6 and aim at:

- quantitatively describing the **slowdown of real-world applications** in the presence of hypervisor-based monitoring
- **comparing** the performance of the two solutions Drakvuf and HVMI
- assessing the **practical implications** of the two different strategies to solve the semantic gap problem

Similarly to what has been done for the micro-benchmark, the macro-benchmark is run on the virtual machines with and without introspection-based monitoring and then the results are compared. These data are presented in Table 7.5 and Table 7.6.

	Avg Time (No Monitoring)	**Avg Time (Monitoring)**	**Slowdown**
Notepad	2129416610.00 ns	8093160260.00 ns	3.80x
Task Manager	12636495600.00 ns	21320145940.00 ns	1.69x
Calculator	2034557210.00 ns	5354781520.00 ns	2.63x
Explorer	3197643660.00 ns	10452039230.00 ns	3.27x
Paint	21356889410.00 ns	58748146880.00 ns	2.75x

Table 7.5: Results of the macro-benchmark on Xen.

	Avg Time (No Monitoring)	**Avg Time (Monitoring)**	**Slowdown**
Notepad	2134102100.00 ns	2156952230.00 ns	1.01x
Task Manager	11938563820.00 ns	12320958160.00 ns	1.03x
Calculator	1955414910.00 ns	2161041210.00 ns	1.11x
Explorer	2950122900.00 ns	3049382470.00 ns	1.03x
Paint	21377903700.00 ns	22958191590.00 ns	1.07x

Table 7.6: Results of the macro-benchmark on Napoca.

In both cases, the "Slowdown" column contains the ratio between the average time taken by the single macro-benchmark to run with monitoring active and the average run time without monitoring. Averages were computed over 5 runs. Also in this case "No Monitoring" means that no introspection-based monitoring solution was not active system-wide, as done with the micro-benchmark experiment.

As reported in Tables 7.6 and 7.5, while the "Slowdown" factor for Drakvuf ranges from **1.69x to 3.80x**, the corresponding value for HVMI is between **1.01x to 1.11x**. The

second range is considerably narrower and its extremes are very close to 1, meaning that running an application with HVMI monitoring against unpacking results in a slowdown considerably lower than Drakvuf.

In our belief, this proves the approach used by HVMI to solve the semantic gap problem in the use case of code unpacking detection is more efficient than the approach implemented by Drakvuf. We tracked down the possible reasons why Drakvuf has a higher overhead than HVMI to several aspects of the behavior of the library. Before presenting them, it is important to stress that we believe that the hooking strategy does not play a major role in Drakvuf increased performance overhead with respect to HVMI since, as proved with the micro-benchmark experiments, the different strategies (altp2m with in-host callbacks and EPT permissions alteration with in-hypervisor callbacks) have similar performance.

First of all, Drakvuf infers the guest execution context from the writes to the CR3 register. As explained in chapter 6, CR3 is written at each guest transition to the kernel or user and upon a context switch. Setting the VMCS to trap on such an event is a possible way of synchronizing with the scheduler of the guest. This approach is easy to implement and does not require deep knowledge of Windows internals, but in past studies, has been documented to heavily impact guest execution because context switches happen very frequently in the operating system and trapping in the hypervisor at every one of them results in excessive exits and stops to the guest execution [21]. HVMI handles process interception differently, in that it tracks Windows activity and calls the hypervisor only upon the scheduling of the monitored process, as explained in chapter 6. Moreover, tracking is realized using detours, stealthy in-guest hooks that only call the hypervisor when strictly needed and that are substantially faster than EPT-based hooks trapping unconditionally in the VMM [31]. On the contrary, Drakvuf makes extensive use of such hooks, leading to each vm-exit suspending the guest.

Another performance problem arises from the approach used by Drakvuf to track process memory accesses. The Codemon plugin discovers executable pages upon the first trap in the hypervisor after an instruction fetch from the page associated with the violation. HVMI avoids this by directly checking in the header of the Windows executable from which the process is spawned which are the executable sections and tracks the guest activity to discover the mappings of such sections to memory frames at process loading time. As a consequence, Drakvuf monitoring results in a certain number of traps at the beginning of the execution of the monitored process, while HVMI does not. This behavior has been empirically observed and confirmed modifying the code of Drakvuf and HVMI to log statistics on process execution and monitoring.

In Codemon, the "trap_counter" variable already mentioned in chapter 6 has been used to count the number of times the execute hook fired during a monitoring session. The variables "trap_switches" and "hooks_counter" have, then, been introduced to count the number of times an execute hook is switched for a write hook and the number of times the hook on MmAccessFault was activated. Table 7.7 reports the statistics just described. The data was collected while monitoring real-world applications during the execution of the macro-benchmark process, i.e. during the execution of the per-process test bench five times in a row.

	Created execute hooks	**Activated execute hooks**	**Trap switches**
Notepad	22121	150	0
Task Manager	89375	657	0
Calculator	3017	5	0
Explorer	78141	534	0
Paint	34425	7399	0

Table 7.7: Statistics on traps in Drakvuf-based monitoring.

Table 7.7 suggests two important facts:

- **Fact 1**: The number of trap switches observed in real-world applications is 0. Separating data and code is of great importance for modern operating systems and application developers and this results in no execute and write access alternation on an executable code page.
- **Fact 2**: The activated execute hooks trapping in the hypervisor is only a small portion of the created ones, suggesting that this method may incur an excessive overhead on guest operations to discover executable pages.

Codemon captures traps on MmAccessFault happening in the context of the process in both kernel and user mode, both concerning the pages allocated inside the virtual address space of the application for mapping its code and data to memory and the entries that are relative to shared code. Even if hooks are not installed on kernel pages and shared libraries, page faults concerning such memory mappings lead to traps in the hypervisor anyway. This results in a certain slowdown of process execution not experienced on HVMI.

For the sake of comparison, also HVMI has been patched to include information logging at certain points of the code of the unpacker.c module to extract statistics on hook management. The extracted statistics are the number of hooked pages and the number of times a write switch was activated and switched with an execute hook. Different from Table 7.7,

the "Activated execute hooks" is not present in Table 7.8 because, as explained, HVMI does not need to trap on the first instruction fetch from a page to discovery executable pages, so it installs directly the write hook on the monitored pages.

	Created execute hooks	**Trap switches**
Notepad	135	0
Task Manager	850	0
Calculator	5	0
Explorer	2730	0
Paint	775	0

Table 7.8: Statistics on traps in HVMI-based monitoring.

Thanks to the way the semantic gap is handled, HVMI can install considerably fewer hooks than Drakvuf overall. No traps in the hypervisor happen at all, because executable pages discovery is not necessary, preventing the slowdown of the first execute accesses on a just-swapped-in page from happening. We believe that from the comparison of the data in Table 7.7 and Table 7.8 it is possible to prove that the best-performing strategy among the ones analyzed to potentially detect code unpacking inside the executable memory of Windows processes is the one adopted by HVMI.

Further considerations on the data are hereby reported.

- Drakvuf monitors dynamically allocated memory regions. Depending on the frequency of memory allocations and accesses to newly allocated memory, traps in the hypervisor upon MmAccessFault invocation may grow substantially. This is one of the reasons why in Drakvuf the number of installed execute hooks for executable page discovery is higher than the number of tracked pages in HVMI. The process Task Manager is one such example.
- The filtering criterion used to avoid installing traps in shared memory locations may be too simplistic. Again, this may explain the substantial number of created hooks, but we believe it is the "price" to pay for not delving too deep into Windows internals to solve the semantic gap. The process Paint is one such example.
- Drakvuf only tracks effectively accessed pages, while HVMI starts monitoring for write accesses pages that may **never** be accessed in the currently running session. For this reason, sometimes the execute hooks trapping in Drakvuf are less than the hooks installed in the executable pages of a process by HVMI. Process Explorer is one such example.

7.4. HyperDbg-based monitoring

The present set of experiments aims to prove that using HyperDbg as a memory monitor is possible from a functional standpoint. We want to show that the script we wrote to implement on HyperDbg a memory monitor for code unpacking detection is functionally equivalent to Codemon and HVMI. We also want to investigate how the lack of flexibility and simplistic nature of trap management facilities in HyperDbg impacts the performance of monitoring more complex applications than the micro-benchmark, as concerns about the efficiency of the tool may rise when its functioning is analyzed.

To accomplish the goals presented in this section, our HyperDbg-based monitoring solution has been tested against the modified version of the micro-benchmark presented in chapter 6. The micro-benchmark has been patched to include variable length shellcode and dynamic memory allocation via VirtualAlloc.
To prove the functional equivalence of the HyperDbg-based monitor to the Drakvuf and HVMI-based monitoring flow, we removed the comment on the prints inside the script reported in Listing 6.14 to log the activity of the hypervisor during the monitoring session. Then we ran the micro-benchmark and captured debugger output to check if memory accesses were correctly tracked. Our objective is to be able to trap in the hypervisor on the first write to a code page, mark it as "dirty" and log this state change. Upon an execute access to the page, another log is produced only if the page is "dirty". To this aim, we launch the micro-benchmark script in the guest and then we start tracking its execution with HyperDbg.

```
1  .\unpacker.exe 5 1
```

Listing 7.5: Prompt of the first micro-benchmark invocation for HyperDbg functional check.

The operation after which monitoring starts is the invocation of VirtualProtect, which is followed by the copy of the shellcode to the page. This triggers the monitoring tool, that makes the page "dirty". The first instruction fetch from the newly allocated memory results in the monitoring infrastructure logging the execute access. Subsequently, write and execute accesses alternate. We wanted to be sure that only the first operation in a sequence of operations of the same type is logged, since it is the one signaling state change in the page and the subsequent ones are not of interest for tracking page state evolution according to the model presented in chapter 2. We proved this is not the case using the patched benchmark supporting variable length shellcode. Running the program with a shellcode length greater than 1 always results in the same behavior on the monitor side, i.e. only the first execute access is tracked. This behavior is depicted in Figure 7.2.

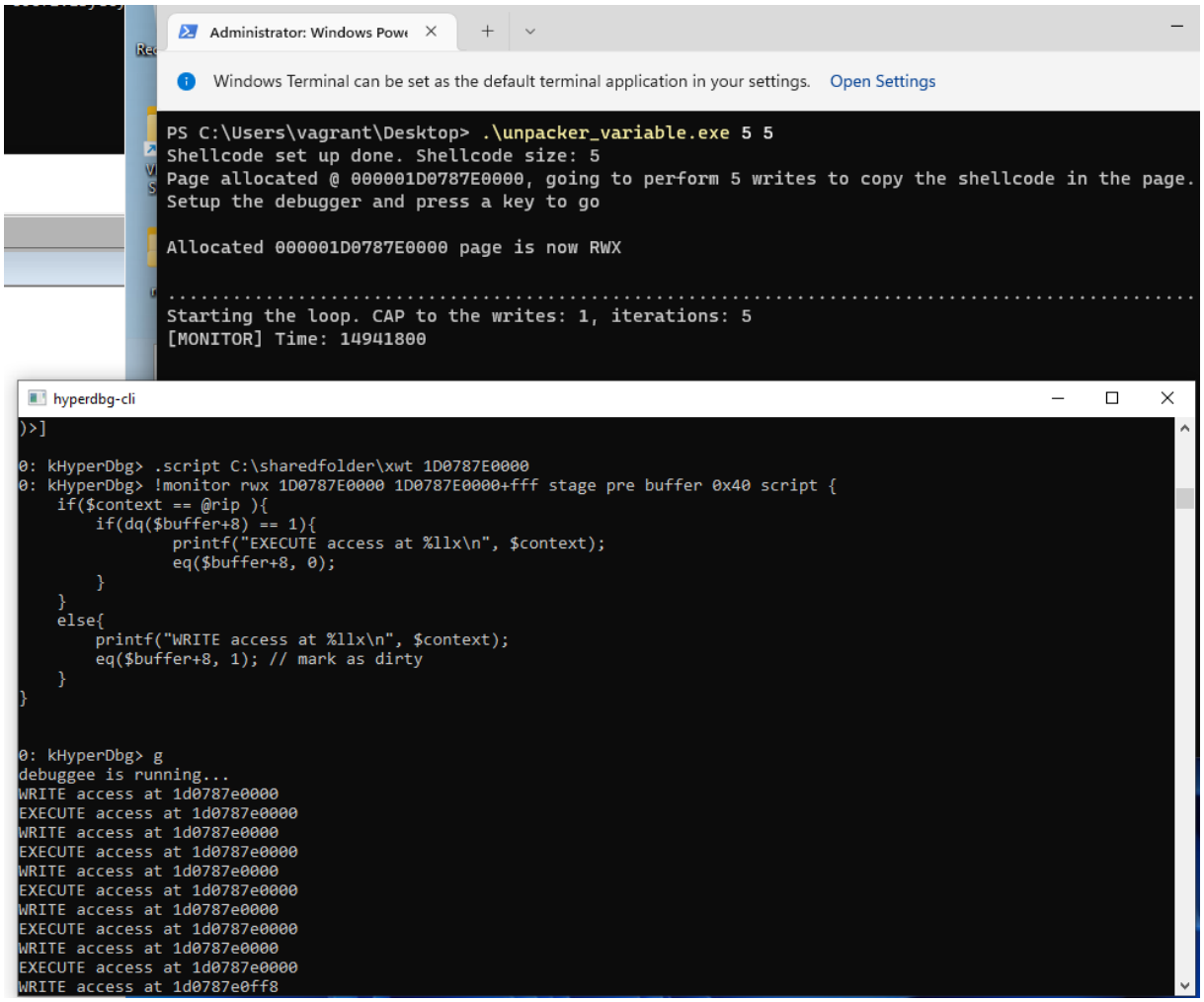

Figure 7.1: Output of one run of the script xwt on HyperDbg.

```
1  .\unpacker.exe 5 4
```

Listing 7.6: Prompt of the second micro-benchmark invocation for HyperDbg functional check with shellcode length 4.

To quantitatively describe the slowdown in the memory operations performed inside the micro-benchmark loop, we run such a program on the HyperDbg virtual machine with and without HyperDbg-based monitoring. It is important to specify that in both cases the HyperDbg hypervisor is active. This is consistent with what has been done in the case of Xen and Napoca. The only difference between the two cases is the activation of introspection-based hooks with the monitoring up and running, according to the implementation described in chapter 6. The patched micro-benchmark program invocation prompt is reported in Listing 7.7.

```
.\unpacker.exe 500000 1
```

Listing 7.7: Prompt to invoke the patched micro-benchmark with shellcode length 1.

As suggested before, the second parameter of the invocation is the length of the shellcode while the first is the number of loop iterations executed by the program. Table 7.9 shows the results of the experiment performed by fixing the length to 1.

	Loop execution time	**Mean per-iteration time**
Monitor OFF	45158459000 ns	117.03 ns
Monitor ON	9031691800 ns	90316.91 ns
Slowdown Factor	771.74x	

Table 7.9: Results of the micro-benchmark test on HyperDbg.

The slowdown factor on one iteration of the loop resulting from the use of HyperDbg as a monitor is **771.74x**, computer as the ratio between the mean per-iteration times in Table 7.9. We believe that the ratio alone cannot prove that HyperDbg is a viable solution for implementing hypervisor-based monitoring.

Trap handling in ring -1 is what HVMI and HyperDbg have in common, but while the debugger exhibits a **771.74x** slowdown factor, HVMI causes a higher performance overhead in the guest (**2212.53x**). We believe the cause of this overhead is the complexity of the callbacks implemented in HVMI to handle traps, which is way higher than the complexity of the simplistic scripts defined in the case of HyperDbg. In HVMI, each callback performs reads and writes in guest registers, accounting, and internal state management actions and logging. Such actions imply a slowdown in guest operations since the guest is stopped during their execution in our setup. Their overhead has been partially investigated in the past by the authors of [50] and [21]. In brief, the greater flexibility and capabilities of HVMI come at the cost of needing to stop the execution of the guest for a longer time than HyperDbg-based monitor.
This being said, we asked ourselves what the impact of the lack of flexibility in HyperDbg described in chapter 6 on the task of monitoring more complex applications is with HyperDbg, strictly focusing on performance. After proving that the monitoring script we wrote grants functional equivalence to HVMI and Drakvuf, we are going to use the extended version of the micro-benchmark to control the size of the shellcode that gets executed in the $\mathbb{W} + \mathbb{X}$ loop. We aimed to understand how the overhead on the guest operations evolves in response to changes in the shellcode length, believing that monitoring overhead data will shed some light on how HyperDbg behaves in real-world contexts.

Three tests have been included in the following set of experiments. In the first one, the length of the shellcode is fixed to **1 byte** (RET only). Subsequently, it is increased to **3 bytes** (2 NOPs, 1 RET) and, lastly, to **5 bytes** (4 NOPS, 1 RET). In every case, 100000 iterations of the $\mathbb{W} + \mathbb{X}$ loop were performed. Data are collected in Table 7.10. The execution times of the loop in the cases of monitoring ON and OFF are reported, together with the slowdown factor computed as their ratio.

	No Monitoring	**Monitoring**	**Slowdown**
Length 1	11703000.00 ns	9031691800.00 ns	771x
Length 3	10332800.00 ns	15467000400.00 ns	1496.88x
Length 5	12184900.00 ns	24960475000.00 ns	2048.48x

Table 7.10: Results of the micro-benchmark on HyperDbg for different shellcode lengths.

According to the data, we can prove that the loop execution slowdown factor grows with the size of the shellcode. This outcome is the consequence of excessive trapping in the hypervisor due to the lack of dynamic trap switching and management facilities at the hypervisor level. As a result, we believe that, at the time of the writing, HyperDbg per se is not usable as a memory monitor in a real-world context. It is our understanding that this does not exclude its use in the future upon the extension of HyperDbg with advanced monitoring facilities to enable dynamic trap management.

7.5. Evading HVMI-based monitoring

When testing Napoca + HVMI against the version of the micro-benchmark patched to include dynamic memory allocation for testing HyperDbg, we found out that the tool is not able to monitor dynamically allocated memory. Hooks are installed at process creation time on memory pages mapping the code section of the binary. HVMI does not hook memory allocation functions for memory access tracing purposes and this implies that whenever new memory areas are dynamically allocated to the program, e.g. using the Windows API VirtualAlloc, no hooks are installed on them. We believe this is a serious limitation that allows potentially malicious programs to perform code unpacking without it being captured by the hypervisor.

To support our hypothesis, the version of the patched micro-benchmark with dynamic memory allocation and variable shellcode length was set as protected against unpacking in HVMI using a prompt similar to the ones already presented in the previous sections. Then, we ran it and checked the logs of the hypervisor. Code pages were successfully

hooked on process start, but there was no sign of hook activation. To go deeper into the question, we modified the code of HVMI to make it output the virtual addresses at which hooks are installed and further information at process termination and we extracted them from the logs of the hypervisor. They also include the number of fired execute hooks, previously used to debug and check on the functioning of our extension code. As can

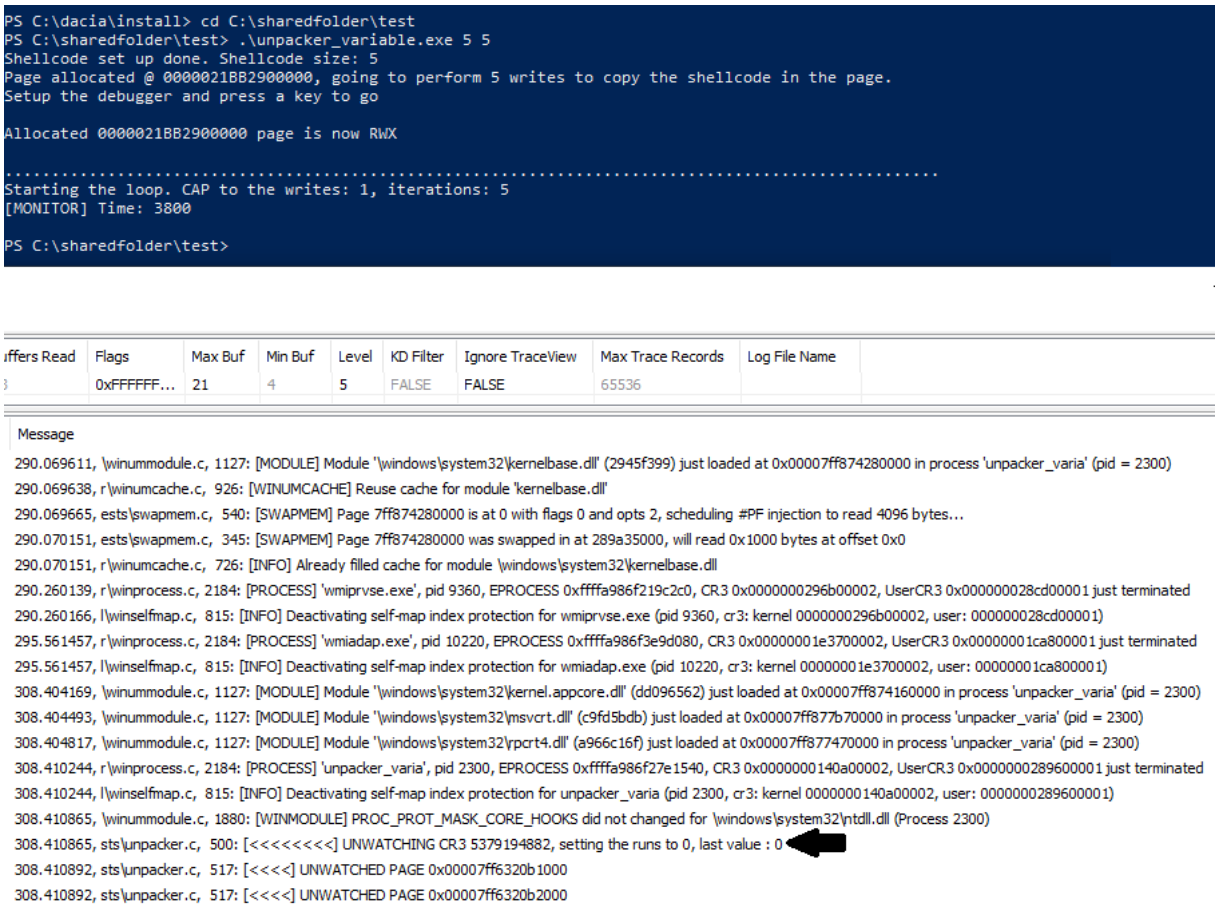

Figure 7.2: Invocation of the patched version of the micro-benchmark on Napoca+HVMI. The dynamically allocated page is not hooked and the execution of the $\mathbb{W} + \mathbb{X}$ loop is not captured, as signaled by the last value 0 of the counter of the execute traps in the page.

be seen in the log, the installed hooks are in the range of contiguous addresses ranging from `0x7ff6320b1000` to `0x7ff63216300`, while the virtual address of the newly allocated memory area is `0x21bb2900000`, which is not included in that range. No hook is installed on that memory location because it is dynamically added to process address space. For this reason, execution and write attempts are not captured.

This is a serious limitation that makes HVMI-based monitoring unsuitable for reliable memory monitoring, as malware can, in principle, unpack in dynamically allocated memory regions, set appropriate permissions, and execute its unpacked code.

7.6. Experiments Challenges

This section describes the principal setbacks encountered while working with the monitoring solutions analyzed in the present thesis.

7.6.1. Drakvuf

Codemon is a plugin written to precisely track memory accesses by guest programs, detect potential unpacking attempts, and optionally dump hidden code that has been unpacked into memory. The design choices at the base of the plugin make it invasive to the monitored guest functioning because it continuously stops its execution, memory accesses to not yet loaded or swapped out pages being a very frequent task. Moreover, page fault injection is a task that can easily lead to instabilities in the guest, as proven by the fact that Codemon was recently extended to include KiSystemServiceHandler hooking. Whenever such a function is called to manage kernel mode exceptions in Windows, Codemon checks the cause of the exception. If it is related to the injection of a page fault in the wrong memory location, Codemon short-circuits the invocation of the routine modifying the value of the current instruction pointer to make it point to the saved RIP on the kernel stack. Nevertheless, kernel-level exceptions happen the same after page fault injection, for reasons we are not able to explain. Finally, during the execution of the tests, we encountered many VM freezes on top of intensive hooking activity as suggested by the logs. We hypothesize that hooking shared memory locations such as DLLs and system routines raises the chance that hooks are not managed properly at the library level, resulting in them being deallocated before the associated callback can tell the hypervisor to resume the guest. These are just hypotheses since we have no way of proving this is the cause of the errant behavior of Codemon. Nonetheless, reconstructing the state of the guest upon its freezing is pretty hard if not impossible since it is neither in a crashed state (according to Xen logs) nor operative. There are no bug checks ongoing and, even if one manages to attach a debugger to the Xen virtual machine leveraging the Xen-emulated serial interface for the VM, the debugger becomes not responsive because it uses in-guest kernel modules that just cannot run.

7.6.2. HVMI

HVMI sometimes obstacles guest operation. The way it is implemented, some components issue hypercalls using breakpoint-based hypervisor trapping as described in chapters 3 and 6. When debugging the user space, debugging traps are intercepted by the hypervisor, which then passes them to the guest injecting a debugging exception via Software

Interrupt number 3, as explained in chapter 2. The trap filtering function associated with breakpoint handling and re-injection often freezes the guest and crashes the hypervisor because it cannot clearly distinguish between breakpoint interrupts associated with in-guest debugging and calls to the hypervisor issued by the HVMI in-guest components. One last word will be expressed on the control one has to HVMI and Napoca. The setup is not debuggable without the use of a serial debugger. Since not every commodity machine has one such interface available for use, this precludes debugging in most setups. Assuming environment differences do not result in excessive behavior drift when moving to a virtual machine-based setup to debug Napoca and HVMI errant behavior, debugging is impossible also when virtualizing Napoca. According to its authors, and practical experience with different hypervisors and virtualized hardware configurations, Napoca cannot boot on a virtual machine. This limits debugging efforts to inserting prints in the hypervisor. Such an option is highly impractical and not always useful when experiencing crashes because logs cannot be accessed, them being not persisted in most cases due to OS freezing. HVMI can be debugged when run in userland, but bugs arising only in the Napoca + HVMI setup are not reproducible.

7.6.3. HyperDbg

The main problems that HyperDbg has are related to the fact that the scripting engine functioning cannot be debugged using the tool. When a hook is triggered, it is impossible to follow closely the execution of the callback in the hypervisor. On the one hand, this is the "price" to pay because of the way code runs in VMX root. On the other, writing callbacks in assembly to strive for maximum flexibility and efficiency is not user-friendly. Trial and error is the only approach allowed in such a case. The major problem associated with the script engine is the fact that conditions can only be defined in assembly. This is complicated to handle and can easily introduce crashes in the guest as practical experience tells us. HyperDbg is not stable enough to deterministically implement complex debugging flows leveraging the hypervisor-based enhancements it is built on. It may happen that, during debugging, the guest freezes. This happens very frequently in user-mode process debugging. The causes are not clear, so we assume this is due to software bugs partly due to the closed nature of Windows, difficult to understand and control at the hypervisor level. Moreover, actions like system call invocation tracing, potentially with some filtering on the process, the core, and the number of the system call may crash the guest or produce undefined behavior that sooner or later leads to instabilities and crashes. Finally, most of the time, the debugged guest crashes when closing the debugging connection and deactivating the hypervisor. This makes testing complicated because of the time needed

to reboot the virtual machine and restore operative conditions with HyperDbg.

8 | Limitations and Future Work

In this chapter, we discuss the identified limitations in the tools and the potential future extensions to this work.

8.1. Limitations

This section summarizes the limitations individuated in the examined tools. Talking of HVMI, it does not handle dynamically allocated memory tracing at all, resulting in code unpacking still being possible also upon protection activation. HVMI resorts to emulation to handle certain types of memory accesses [16]. Emulating in the context of memory hooking allows the monitoring solution to use a single EPT view for a monitored VM, but it may represent a problem, [33] and it has proven not to be the fastest alternative overall.

Talking of the Drakvuf plugin Codemon, this latter causes instabilities in the guest. They rely on information made available upon reverse engineering attempts on Windows, assumptions which may not cover all the possible corner cases in the execution of an operating system and workarounds whose guarantees are questionable, like kernel exception handler hooking and exception ignoring. We could not track down the causes of all the problems we had, but using Codemon crashes are very frequent, some applications are impossible to monitor and memory usage in the host may become excessive (over 100MB in some test benches).

The fact that Drakvuf still lacks solid introspection debugging facilities makes handling instabilities complicated. On some events, such as a page fault erroneously injected into the guest, the guest crashes or just enters an erroneous state. Having the possibility to perform debugging using the introspection library itself would help investigate the causes of such crashes or errors created in the guest.

From our empirical experience, we understand that some crashes are due to inconsistent reading in LibVMI. This latter library sometimes returns stale data when reading the guest memory. In such case, manual LibVMI cache flushes are the only way to cope

with the anomalies but they have a certain performance impact on introspection. Careful debugging is needed to spot cached stale data reads. Libvmi caching system should be better integrated with the monitored OS, as happens with HVMI.

None of the used tools is able to effectively track shared code execution. Code sharing in Windows revolves around the concept of DLL (Dynamically Linked Library). DLL pages are mapped inside the Virtual Address space of each process using such libraries. HVMI is able to detect this and purposely avoids tracing such code to keep the number of vmexits low. Drakvuf can track them but practical experience suggests that trapping in the hypervisor is too frequent and ends up hurting guest performance.

Last, HyperDbg is a debugger, it does not implement trap switching, resulting in the impossibility of removing hooks in the callbacks running in VMX root mode.

8.2. Future work

We believe future works should take into exam existing VMI solutions and try to limit exits to the hypervisor as much as possible. Exits impact heavily on the guest performance [41] and in-guest hypervisor-protected monitoring code may be used to limit them when implementing monitoring actions [31]. HVMI already uses advanced hardware features to support in-guest detours management and violations handling but, at least in the latter case, they do not seem to improve the performance of code unpacking detection at the hypervisor. Future research should focus on retrofitting existing systems using such features for efficient in-guest hypervisor-protected memory tracing for unpacking detection, possibly integrating hooks to such an in-guest component in OS paging routines. HyperDbg should be extended to introduce more complex commands into the scripting engine to dynamically manage hooks. Signature-based malware detection schemes can be integrated at the hypervisor level to help detect known malwares, upon potential unpacking detection. Lastly, future research work should investigate the stealthiness of hypervisor-based monitoring solutions. Some of the changes done to the guest are potentially invasive, so further research is needed to understand their impact on guest execution. Moreover, many solutions use emulation to handle violations in the hypervisor, which proved to be dangerous [33]. Future research should examine the implications of such design choices.

9 | Conclusions

The present thesis tries to shed some light on the performance degradation in the execution of processes in a virtual machine with hypervisor-based monitoring for code unpacking detection. Packing behavior is exhibited by the so-called packer malware. Packed/compressed malicious programs have to be unpacked/decompressed in memory before they can execute on the target system. For this reason, detecting unpacking can potentially unveil the presence of malicious activity inside a running system.

In this thesis we proposed a simplistic model for code unpacking and, based on this, we proposed a model of the memory access tracking task to detect unpacking at runtime. Based on it, we tested existing solutions for execution monitoring to understand their impact on the performance of running systems. Particularly focusing on memory operations, the solutions analyzed in the present thesis sit at the hypervisor level. Since most malware tries to evade detection measures running at the same privilege level as theirs, moving the protection layer to a more privileged position may help counteract evasive behavior and have good visibility on the system.

This opens up new challenges related to the way software execution at less privileged layers should be traced for fine-grained control, coping with the *semantic gap* existing between different privilege spheres. In the case of hypervisor-based monitoring, the semantic gap is solved using virtual machine introspection for state reconstruction. To actively trace the operations of a virtualized operating system, monitoring solutions make extensive use of hooks, implemented leveraging modern hardware advanced features and triggered upon the verification of the event of interest.

We studied two possible approaches to memory hooking, and for solving the semantic gap problem. We restricted the focus to capturing code execution attempts on just written memory locations. The libraries implementing these approaches, Drakvuf and HVMI, have been tested on the Xen and the Napoca hypervisor, respectively. Our experiments were based on a synthetic micro-benchmark to assess the performance impact on memory operations associated with memory hooking and on a macro-benchmark to evaluate the performance degradation incurred by the execution of real-world programs being moni-

tored to detect unpacking.

From our analysis, the two tested strategies (**EPT permissions alteration** on HVMI and **altp2m** on Drakvuf) result in similar performance degradation on memory accesses when tracking sequences of accesses of interest. In particular, with Drakvuf hooks memory accesses experience a **2120x** slowdown, while, for HVMI, the slowdown factor is **2212x**. We concluded this datum is not enough to state that altp2m is the most suitable alternative for memory hooking from a performance standpoint, as the two computed factors are very close numerically.

The distinction between the two approaches is in the way they influence the implementation of hypervisor-based monitoring solutions. In the case of HVMI, removing permissions from EPT without duplicated memory views often requires instruction emulation [16]. On many hypervisors this is a problematic situation [33]. Altp2m enables hypervisor-based monitoring solutions to restart guest execution without emulation, and subsequently restore restrictions on guest operations. Since page table switching introduces overhead, we studied the cost in terms of time of alternating page tables and found out that, on average, switching the page table pointer for a single vCPU costs **44** times the execution of a write and an instruction fetch in the virtual machine.

We, then, explored the overhead of hypervisor-based monitoring on real-world applications and pointed out that the most efficient approach among the tested ones for solving the semantic gap to detect unpacking at the hypervisor level is implemented by HVMI. This latter library hooks internal, sometimes undocumented, software routines and data structure in the virtualized operating system to closely track process initialization, virtual address space allocation, and memory accesses to precisely *discover executable pages* inside a process. This approach requires in-depth knowledge of Windows operating system internals, but results in monitored programs nearly running at native speed. The worst slowdown factor observed is **1.11x**. Based on this, we conclude that the HVMI approach to hypervisor-based code unpacking detection is suitable for **real-time** monitoring.

The approach implemented by Drakvuf does not require deep knowledge of the internals of the monitored operating system, making it simpler to implement than HVMI but also more impacting on the guest execution. For this introspection library, the worst observed slowdown on the execution of a program is **3.80x**. Nonetheless, Drakvuf handles a corner case not correctly handled by HVMI when monitoring against unpacking. We discussed it.

In the last part of the thesis, we tried to retrofit the HyperDbg hypervisor-based debugger as a memory monitor for unpacking detection. We proved that, from a functional

standpoint, such a debugger behaves equivalently to the other monitoring solutions when tracing the execution of the micro-benchmark written to assess the performance overhead on memory operations of monitoring tools. We found out that the performance penalty on memory access tracing is lower than in the other cases with a slowdown factor of **771x**, considerably lower than the **2000x** of Drakvuf and HVMI.

Nonetheless, due to the design of the tool, it is not usable to monitor applications that are more complex than the micro-benchmark at hand. We proved this by testing the performance of the debugger against a modified version of the micro-benchmark with variable shellcode length to understand how the massive use of single-stepping influences the performance of the monitored system. We showed that the performance overhead on memory accesses grows with the number of instructions fetches in the monitored location. We concluded that this makes the current version of HyperDbg unsuitable for scalable memory monitoring on applications.

In the following, we provide some guidelines for the implementation of efficient hypervisor-based monitoring for code unpacking detection solutions.

1. The monitoring solution should be built on a **deep knowledge** of the monitored operating system internals. This helps to identify the most important points in the operating system code that should be hooked and monitored to efficiently solve the semantic gap problem. Over-general solutions produce excessive overhead.

2. **Context-sensitive** process-based monitoring should be implemented by exploiting preliminary information about the executable. Using data extracted from the program it is possible to gain insights into the memory layout of the monitored process at run-time, which has proved to be more efficient than discovering memory locations of interest at access time trapping in the hypervisor.

3. **Hooking** must be implemented with detours when possible, especially when hooking potentially frequent events (e.g. page faults). Hypercalls in the detour handling code must be limited as much as possible to reduce VM exits to a bare minimum as they have been shown to harm guest performance [41].

4. **Dynamic memory allocations** and, in general, state changes in the memory layout of a running process have to be closely tracked, otherwise evading the monitoring solutions is trivial.

Bibliography

[1] C++ chrono. URL https://en.cppreference.com/w/cpp/chrono.

[2] introduction to hvmi, . URL https://bitdefender.github.io/hvmiblog/introspection/2020/07/30/introduction.html.

[3] unpacker.c module in hvmi, . URL https://hvmi.readthedocs.io/en/latest/_static/doxygen/html/unpacker_8c_source.html.

[4] HyperDbg docs. URL https://docs.hyperdbg.org/.

[5] *Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4*. URL https://cdrdv2.intel.com/v1/dl/getContent/671200.

[6] Libvmi docs. URL https://libvmi.com/docs/.

[7] linux debugging docs. URL https://docs.kernel.org/dev-tools/gdb-kernel-debugging.html.

[8] Design of the monitor command on hyperdbg. URL https://docs.hyperdbg.org/design/features/vmm-module/design-of-monitor.

[9] Windows reverse engineering notes. URL https://undocumented.ntinternals.net/index.html?page=UserMode%2FUndocumented%20Functions%2FMemory%20Management%2FVirtual%20Memory%2FNtProtectVirtualMemory.html.

[10] Python time documentation. URL https://docs.python.org/3/library/time.html.

[11] Xen hypercalls. URL https://xenbits.xen.org/docs/unstable/hypercall/x86_64/index.html.

[12] A. Abdelraoof, B. Taubmann, T. Dangl, and H. P. Reiser. Introspect virtual machines like it is the linux kernel! In L. Bilge, L. Cavallaro, G. Pellegrino, and N. Neves, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 258–277, Cham, 2021. Springer International Publishing. ISBN 978-3-030-80825-9.

[13] O. A. Aslan and R. Samet. A comprehensive review on malware detection approaches. *IEEE Access*, 8:6249–6271, 2020. doi: 10.1109/ACCESS.2019.2963724.

[14] E. Bauman, G. Ayoade, and Z. Lin. A survey on hypervisor-based monitoring: Approaches, applications, and evolutions. *ACM Comput. Surv.*, 48(1), aug 2015. ISSN 0360-0300. doi: 10.1145/2775111. URL https://doi.org/10.1145/2775111.

[15] BitDefender. Hvmi: activation and protection options. URL https://hvmi.readthedocs.io/en/latest/chapters/2-activation-and-protection-options.html.

[16] Bitdefender. URL https://github.com/bitdefender/hvmi/blob/master/introcore/src/guests/callbacks.c#L1360.

[17] BitDefender. Hvmi source code, . URL https://hvmi.readthedocs.io/en/latest/_static/doxygen/html/winummodule_8c_source.html#l01851.

[18] BitDefender, . URL https://hvmi.readthedocs.io/en/latest/_static/doxygen/html/introapi_8c_source.html#l00492.

[19] BitDefender. Hvmi docs. Blog Post, 2020. URL https://hvmi.readthedocs.io/en/latest/.

[20] E. Bugnion, S. Devine, M. Rosenblum, J. Sugerman, and E. Y. Wang. Bringing virtualization to the x86 architecture with the original vmware workstation. *ACM Trans. Comput. Syst.*, 30(4), nov 2012. ISSN 0734-2071. doi: 10.1145/2382553.2382554. URL https://doi.org/10.1145/2382553.2382554.

[21] T. Dangl, B. Taubmann, and H. P. Reiser. Rapidvmi: Fast and multi-core aware active virtual machine introspection. In *Proceedings of the 16th International Conference on Availability, Reliability and Security*, ARES '21, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450390514. doi: 10.1145/3465481.3465752. URL https://doi.org/10.1145/3465481.3465752.

[22] A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, page 5162, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781595938107. doi: 10.1145/1455770.1455779. URL https://doi.org/10.1145/1455770.1455779.

[23] H. foundation. Hyperdbg - events. Blog Post, 2023. URL https://docs.hyperdbg.org/design/debugger-internals/events.

[24] N. Galloro, M. Polino, M. Carminati, A. Continella, and S. Zanero. A systematical

and longitudinal study of evasive behaviors in windows malware. *Computers and Security*, 113:102550, 2022. ISSN 0167-4048. doi: https://doi.org/10.1016/j.cose.2021.102550. URL https://www.sciencedirect.com/science/article/pii/S0167404821003746.

[25] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Network and Distributed System Security Symposium*, 2003. URL https://api.semanticscholar.org/CorpusID:6136159.

[26] T. Garfinkel, K. Adams, A. Warfield, and J. Franklin. Compatibility is not transparency: Vmm detection myths and realities. In *USENIX Workshop on Hot Topics in Operating Systems*, 2007. URL https://api.semanticscholar.org/CorpusID:2768904.

[27] T. G. D. Gruss. Cloud operating systems notes. URL https://www.iaik.tugraz.at/wp-content/uploads/2022/09/slides-6.pdf.

[28] Y. Hebbal, S. Laniepce, and J.-M. Menaud. Virtual machine introspection: Techniques and applications. In *2015 10th International Conference on Availability, Reliability and Security*, pages 676–685, 2015. doi: 10.1109/ARES.2015.43.

[29] ired.team. System service descriptor table. URL https://www.ired.team/miscellaneous-reversing-forensics/windows-kernel-internals/glimpse-into-ssdt-in-windows-x64-kernel.

[30] M. G. Kang, P. Poosankam, and H. Yin. Renovo: a hidden code extractor for packed executables. In *Proceedings of the 2007 ACM Workshop on Recurring Malcode*, WORM '07, page 4653, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595938862. doi: 10.1145/1314389.1314399. URL https://doi.org/10.1145/1314389.1314399.

[31] M. S. Karvandi, M. Gholamrezaei, S. Khalaj Monfared, S. Meghdadizanjani, B. Abbassi, A. Amini, R. Mortazavi, S. Gorgin, D. Rahmati, and M. Schwarz. Hyperdbg: Reinventing hardware-assisted debugging. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 1709–1723, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450394505. doi: 10.1145/3548606.3560649. URL https://doi.org/10.1145/3548606.3560649.

[32] Y. Kawakoya, M. Iwamura, and M. Itoh. Memory behaviorbased automatic malware unpacking in stealth debugging environment. In *2010 5th International Conference*

on Malicious and Unwanted Software, pages 39–46, 2010. doi: 10.1109/MALWARE.2010.5665794.

[33] T. K. Lengyel. Xen altp2m. URL `https://xenproject.org/2016/04/13/stealthy-monitoring-with-xen-altp2m/`.

[34] T. K. Lengyel, S. Maresca, B. D. Payne, G. D. Webster, S. Vogl, and A. Kiayias. Scalability, fidelity and stealth in the drakvuf dynamic malware analysis system. In *Proceedings of the 30th Annual Computer Security Applications Conference*, ACSAC '14, page 386–395, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450330053. doi: 10.1145/2664243.2664252. URL `https://doi.org/10.1145/2664243.2664252`.

[35] R. S. Leon, M. Kiperberg, A. A. Leon Zabag, and N. J. Zaidenberg. Hypervisor-assisted dynamic malware analysis. *Cybersecurity*, 4(1):1–14, 2021.

[36] L. Litty, H. A. Lagar-Cavilla, and D. Lie. Hypervisor support for identifying covertly executing binaries. In *USENIX Security Symposium*, volume 22, page 70, 2008.

[37] L. Martignoni, M. Christodorescu, and S. Jha. Omniunpack: Fast, generic, and safe unpacking of malware. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, pages 431–441, 2007. doi: 10.1109/ACSAC.2007.15.

[38] Microsoft. Virtualalloc function documentation, . URL `https://learn.microsoft.com/en-gb/windows/win32/api/memoryapi/nf-memoryapi-virtualalloc`.

[39] Microsoft. *Microsoft Windows UIA documentation*, . URL `https://docs.microsoft.com/en-us/windows/win32/winauto/uiauto-overview`.

[40] P. Mishra, P. Aggarwal, A. Vidyarthi, P. Singh, B. Khan, H. H. Alhelou, and P. Siano. Vmshield: Memory introspection-based malware detection to secure cloud-based services against stealthy attacks. *IEEE Transactions on Industrial Informatics*, 17(10): 6754–6764, 2021. doi: 10.1109/TII.2020.3048791.

[41] V. Narayanan and A. Burtsev. The opportunities and limitations of extended page table switching for fine-grained isolation. *IEEE Security & Privacy*, 21(3):16–26, 2023. doi: 10.1109/MSEC.2023.3251385.

[42] O. Or-Meir, N. Nissim, Y. Elovici, and L. Rokach. Dynamic malware analysis in the modern era—a state of the art survey. *ACM Comput. Surv.*, 52(5), sep 2019. ISSN 0360-0300. doi: 10.1145/3329786. URL `https://doi.org/10.1145/3329786`.

[43] O. Or-Meir, N. Nissim, Y. Elovici, and L. Rokach. Dynamic malware analysis in the

modern era—a state of the art survey. *ACM Comput. Surv.*, 52(5), sep 2019. ISSN 0360-0300. doi: 10.1145/3329786. URL https://doi.org/10.1145/3329786.

[44] B. D. Payne. Simplifying virtual machine introspection using libvmi. 2012. URL https://api.semanticscholar.org/CorpusID:59261468.

[45] G. J. Popek and R. P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, jul 1974. ISSN 0001-0782. doi: 10.1145/361011.361073. URL https://doi.org/10.1145/361011.361073.

[46] X. Project. Vmi on xen. URL https://wiki.xenproject.org/wiki/Virtual_Machine_Introspection.

[47] Pywinauto. Pywinauto documentation, 2021. URL https://pywinauto.readthedocs.io/en/latest/.

[48] R. Riley, X. Jiang, and D. Xu. Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing. In R. Lippmann, E. Kirda, and A. Trachtenberg, editors, *Recent Advances in Intrusion Detection*, pages 1–20, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-87403-4.

[49] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee. Polyunpack: Automating the hidden-code extraction of unpack-executing malware. In *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, pages 289–300, 2006. doi: 10.1109/ACSAC.2006.38.

[50] S. Suneja, C. Isci, E. de Lara, and V. Bala. Exploring vm introspection: Techniques and trade-offs. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '15, page 133–146, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450334501. doi: 10.1145/2731186.2731196. URL https://doi.org/10.1145/2731186.2731196.

[51] S. Tanda. Hypervisor-101-in-rust. URL https://github.com/tandasat/Hypervisor-101-in-Rust.

[52] X. Ugarte-Pedrero, D. Balzarotti, I. Santos, and P. G. Bringas. Sok: Deep packer inspection: A longitudinal study of the complexity of run-time packers. In *2015 IEEE Symposium on Security and Privacy*, pages 659–673, 2015. doi: 10.1109/SP.2015.46.

[53] R. Uhlig, G. Neiger, D. Rodgers, A. Santoni, F. Martins, A. Anderson, S. Bennett, A. Kagi, F. Leung, and L. Smith. Intel virtualization technology. *Computer*, 38(5): 48–56, 2005. doi: 10.1109/MC.2005.163.

[54] Wikipedia. W xor x. URL https://en.wikipedia.org/wiki/W%5.

[55] C. Willems, R. Hund, and T. Holz. Tr-hgi-2012-002 cxpinspector : Hypervisor-based , hardware-assisted system monitoring. 2012. URL https://api.semanticscholar.org/CorpusID:53669514.

[56] Xen. Xen features. URL https://github.com/xen-project/xen/blob/master/SUPPORT.md.

[57] H. Xiong, Z. Liu, W. Xu, and S. Jiao. Libvmi: A library for bridging the semantic gap between guest os and vmm. In *2012 IEEE 12th International Conference on Computer and Information Technology*, pages 549–556, 2012. doi: 10.1109/CIT.2012.119.

[58] V. Zaccaria. Advanced operating systems notes on virtualization. URL https://drive.google.com/open?id=19FQEPwHOCi_bqglzOGx_0Gm5MVjEngzi&usp=drive_fs.

[59] Y. Zhang. Memory virtualization. URL https://cseweb.ucsd.edu/~yiying/cse291j-winter20/reading/Virtualize-Memory.pdf.

A | Appendix A

```
int main(int argc, const char * argv[])
{
    DWORD oldp;
    int its = argc > 1 ? atoi(argv[1]) : 10;

    int shellcode_size = -1;
    if (argc > 2) {
        shellcode_size = atoi(argv[2]);
    }
    else {
        shellcode_size = 1;
    }
    char* shellcode = new char[shellcode_size];
    for (int i = 0; i < shellcode_size-1; i++) {
        shellcode[i] = 0x90;
    }
    shellcode[shellcode_size-1] = 0xC3; // ret instruction
    printf("Shellcode set up done. Shellcode size: %d\n", shellcode_size
   );

    void* p = VirtualAlloc(NULL, 4096, MEM_COMMIT|MEM_RESERVE,
   PAGE_READWRITE) ;
    if (!p) {
    printf("Failed to allocate memory\n");
    return -1;
  }
    char* ptr = reinterpret_cast<char*>(p);

    printf("Page allocated @ %p, going to perform
            %d writes to copy the shellcode in the page.\n", p,
   shellcode_size);
    for (int i = 0; i < shellcode_size; i++) {
        ptr[i] = shellcode[i];
    }

```

```
33      if (VirtualProtect(p, 4096, PAGE_EXECUTE_READWRITE, &oldp))
34          printf("Allocated %p page is now RWX\n", p);
35
36      int CAP = argc>3? atoi(argv[2]) : 1;
37
38      void (*f) () = (void (*)())p; // cast page pointer to a function
        pointer
39
40      printf("Starting the loop. CAP to the writes: %d, iterations: %d\n",
        CAP, its);
41      std::chrono::time_point<std::chrono::high_resolution_clock> start,
        end;
42
43      // W/X loop
44      start = std::chrono::high_resolution_clock().now();
45      while (its > 0) {
46          for (int i = 0; i < CAP; i++) {
47              ptr[i] = shellcode[i];
48          }
49          f();
50          its--;
51      }
52      end = std::chrono::high_resolution_clock().now();
53      // end loop
54
55      auto time = std::chrono::duration_cast<std::chrono::nanoseconds>(end
        - start);
56      printf("[MONITOR] Time: %llu \n", time.count());
57      VirtualFree(p);
58      return 0;
59  }
```

Listing A.1: Main function of unpacker.exe, extended version.

```
1  #include <xenctrl.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  int SIZE = 2048; // bytes
6  int ENTRY_SIZE = 8;
7  int ENTRIES = 256;
8
9  int main(int argc, const char *argv[])
10 {
11     xc_interface *xch = xc_interface_open(NULL, NULL,
```

```
    XC_OPENFLAG_NON_REENTRANT);
12   int domid = atoi(argv[1]);
13   int rc = -1; //when negative, we have a problem
14
15   // check if alp2m is enabled
16   bool b = true;
17
18   rc = xc_altp2m_get_domain_state(xch, domid, &b);
19   if(rc<0)
20   {
21       fprintf(stderr, "Error\n");
22       goto exit;
23   }
24
25   else printf(b ? "altp2m on\n" : "altp2m off\n");
26
27   // enable altp2m
28   if (!b) // B FALSE WHEN ALTP2M OFF
29   {
30       rc = xc_altp2m_set_domain_state(xch, domid, true);
31       if(rc<0)
32           {
33               fprintf(stderr, "Error in setting altp2m on.\n");
34               goto exit;
35           }
36
37   }
38
39   printf("creating new view with guest original permissions...\n");
40
41   /** EPT PERMISSIONS
42   PERMISSIONS xen_memaccess_t:
43   XENMEM_access_n,
44   XENMEM_access_r,
45   XENMEM_access_w,
46   XENMEM_access_rw,
47   XENMEM_access_x,
48   XENMEM_access_rx,
49   XENMEM_access_wx,
50   XENMEM_access_rwx,
51   *
52    * Page starts off as r-x, but automatically
53    * change to r-w on a write
54    *
```

```
55      XENMEM_access_rx2rw,
56      /*
57       * Log access: starts off as n, automatically
58       * goes to rwx, generating an event without
59       * pausing the vcpu
60       *
61      XENMEM_access_n2rwx,
62      * Take the domain default
63      XENMEM_access_default
64  */
65
66      // create view
67      short int original_view_id = -1, newlyallocv = -1;
68      rc = xc_altp2m_create_view(xch, domid, XENMEM_access_default, &
    newlyallocv);
69      if(rc<0){
70          fprintf(stderr, "Error creating a new view... \n");
71          goto exit;
72      }
73
74      else printf("Created new view with id: %d\n", newlyallocv);
75
76      // get current idx
77      int vcpu_id = 0; // the virtual machine has a single vcpu
78      short int altp2m_idx = -1;
79      if (xc_altp2m_get_vcpu_p2m_idx(xch, domid, vcpu_id, &altp2m_idx) <
    0)
80          fprintf(stderr, "Error\n");
81      else
82          printf("This is the current view index %d\n", altp2m_idx);
83
84      // switch to the new view
85      if (xc_altp2m_switch_to_view(xch, domid, newlyallocv) < 0)
86          {
87              fprintf(stderr, "Error switching to the new view with ID: %d
    \n", newlyallocv);
88              // should disable altp2m here ...
89              goto exit;
90          }
91      else printf("Switched to view %d\n", newlyallocv);
92
93      original_view_id = 0; // original view
94      if (xc_altp2m_switch_to_view(xch, domid, 0) < 0)
95          fprintf(stderr, "Error switching to the new view with ID: %d\n",
```

```
    original_view_id);
96      printf("Switched to view %d\n", original_view_id);
97
98      if (xc_altp2m_destroy_view(xch, domid, newlyallocv))
99          fprintf(stderr, "Error destroying the view\n");
100     else
101         printf("View %d destroyed\n", newlyallocv);
102 exit:
103     xc_interface_close(xch);
104 }
```

Listing A.2: Main function of altp2m-bench.c, benchmark for altp2m.

List of Figures

Listings

List of Tables

List of Symbols

Variable	Description
$\mathbb{W} + \mathbb{X}$	execute+write loop

Acronyms

API Application Programming Interface. 11–15, 19, 21, 24–27, 41–43, 45, 49, 52, 55–57, 59, 64–66, 68, 70

CLI Command Line Interface. 47

CPU Central Processing Unit. 9, 17, 40, 47, 80–82

CR3 Control Register 3. 12, 20, 27, 29, 60, 63, 64, 66, 73, 88

DLL Dynamic-link library. 46, 62, 74, 75, 119

EPT Extended Page Tables. 9, 10, 12, 15, 21, 24–27, 29, 32, 39, 40, 47, 48, 79, 82, 86, 88, 99, 102

EPTP Extended Page Tables Pointer. 10, 37, 61

GFN Guest Frame Number. 64

GUI Graphical User Interface. 70–72

HVMI Hypervisor Memory Introspection. i, iii, 2, 3, 32, 33, 37, 53, 55, 60, 65–69, 74–76, 79, 80, 82, 84–91, 93–97, 99–103, 117, 119, 121

IDT Interrupt Descriptor Table. 23, 25

IOMMU Input/Output Memory Management Unit. 26

MMU Memory Management Unit. 20, 23

MSR Model Specific Register. 11, 12, 22, 23

MTF Monitor Trap Flag. 60, 65, 69

NX Non Executable. 23, 24

OS Operating System. 5, 7, 13–16, 19, 21, 22, 24, 26, 29, 31, 32, 44, 45, 84, 97, 100

SDM Software Development Manual. 10, 121

SLAT Second Layer Address Translation. 7, 9, 26, 27, 39, 48

TLB Translation Lookaside Buffer. 16, 17

TSC Time Stamp Counter. 12, 17, 22

UEFI Unified Extended Firmware Interface. 26

VE Virtualization Exception. 85, 86, 119, 121

VM Virtual Machine. vi, 10, 21, 22, 24–30, 80, 81, 85, 88, 94–97, 117

VMCS Virtual Machine Control Structure. 8–12, 47

VMI Virtual Machine Introspection. 2, 11, 13–15, 82, 99, 100

VMM Virtual Machine Monitor. 7–13, 17, 88