RUHR-UNIVERSITÄT BOCHUM

Horst Görtz Institute for IT Security

Technical Report TR-HGI-2012-002

CXPInspector: Hypervisor-Based, Hardware-Assisted System Monitoring

Carsten Willems, Ralf Hund, Thorsten Holz

Chair for Systems Security

Ruhr-Universität Bochum
Horst Görtz Institute for IT Security
D-44780 Bochum, Germany

TR-HGI-2012-002
November 26, 2012

CXPInspector: Hypervisor-Based, Hardware-Assisted System Monitoring

Carsten Willems, Ralf Hund, Thorsten Holz

Abstract

In the last few years, many different techniques were introduced to analyze a given binary executable. Most of these techniques take advantage of *Virtual Machine Introspection* (VMI), the process of analyzing the state of a virtual machine from the outside. On the one hand, many approaches are based on system emulators which enable a tight control over the program execution. Unfortunately, such approaches typically induce a huge performance overhead. On the other hand, there are approaches based on hypervisors. Early implementations were hampered by the missing virtualizability of the x86 instruction set architecture: since the memory management unit (MMU) itself was not virtualized, memory separation needed to be enforced in software with the help of so called *shadow page tables*, an approach that again induced performance overhead. However, processor vendors have recently added hardware support for MMU virtualization and modern CPUs offer so called *Two-Dimensional Paging* to overcome such performance bottlenecks.

In this paper, we study how this processor feature can be utilized to implement a binary analysis framework. More specifically, we introduce an approach to monitor code execution based on the concept of *Currently eXecutable Pages* (CXP), i.e., we precisely control which memory pages are currently executable to enable the interception of intermodular function calls and their corresponding returns. When an interception occurs, we apply VMI to deduce runtime information such as function parameters. To demonstrate the practical feasibility of the proposed approach, we implemented CXPInspector, a framework for binary analysis on 64-bit machines and Windows 7. In several case studies, we present different application scenarios for CXPInspector: first, we demonstrate how the kernel rootkit TDSS/TDL4 can be analyzed in an automated way. Second, we show how our tool can be used for transparent and efficient performance profiling in a case study with the Apache webserver.

1 Introduction

1.1 Motivation

Virtual Machine Introspection (VMI) is the process of analyzing the state of a virtual machine (VM) from the underlying *virtual machine monitor* (VMM) [1]. The basic idea is to observe the guest VM's memory and deduce what actions a specific processes or even the OS kernel in the VM performs. In practice, the analysis process is complicated due to the so called *semantic gap* problem [2] since the VMM needs to interpret the "raw" memory view obtained by inspecting the VM's memory. However, in recent years several techniques have been developed to overcome this problem in an automated way [3,4]. VMI is a powerful technique with different application domains such as intrusion detection [1, 5, 6], analysis of malicious software [7–10], and related security applications [11–14]. Especially the analysis of malicious software is an important application for VMI, mainly since this technique offers several appealing features relevant for the analysis process such as for example:

- *transparency*: the analysis is invisible for the malware since the instrumentation is performed "out-of-guest".
- *isolation*: the analysis component is isolated from malware and thus the adversary cannot influence it.

- *soundness*: every instruction performed by the malware can be analyzed and intercepted, enabling a fine granular view of the malware's behavior.

Due to the practical relevance, many different malware analysis environments have been developed in the past years [7, 9, 10, 15–20] and lots of them take advantage of VMI in one way or the other. Many of these systems use software-based system emulators such as QEMU to implement the hypervisor and, hence, suffer a huge performance penalty. Also early attempts to utilize hardware-based hypervisors for that purpose were hampered by the missing virtualizability of the x86 instruction set architecture at that time [21]. A substantial performance overhead was caused by the fact that the *memory management unit* (MMU) itself was not virtualized. Due to the missing hardware support to separate the physical memory of the individual guest VMs, hypervisors had to emulate it with the help of so called *shadow page tables* (SPTs) in software.

To solve this problem, processor vendors such as Intel and AMD have recently added hardware support for MMU virtualization in their latest CPUs via so called *Two-Dimensional Paging* (TDP). More specifically, another step at the end of the MMU translation process is added (i.e., from a guest physical address to a host physical address). This significantly improves performance and enables new application scenarios that could be utilized to improve existing monitoring systems.

The main idea behind our work is to take advantage of this TDP feature to divide the guest VM's memory into two partitions, from which one is executable and the other is not. With that splitting in place, we are able to efficiently detect the control flow transitions between certain parts of the memory, i.e. between certain user or kernel modules. The performance advantage is reached by letting the VM execute without further interruption, as long as only instructions from the currently executable memory are executed. By specifically selecting this memory, we are able to monitor the behavior of a system with different scope and variable granularity.

In the following, we focus on the idea to utilize the TDP feature to implement an analysis framework capable of monitoring in detail the behavior of an individual process or even a full virtual machine. Before providing an overview of our approach, we first describe the architectural requirements and design goals.

1.2 Design Goals

Compared to previous work, our approach has some fundamental differences, especially in the way we interact — or better: do *not* interact — with the monitored system. Our design principles are: *transparency*, *OS independence*, *soundness*, *flexibility*, *isolation*, and *timeliness*. We briefly explain each of them to motivate our design choices and explain how we improve existing methods.

Transparency A fundamental requirement when analyzing malware is transparency, especially since malicious software often tries to detect analysis environments to conceal its real purpose and behavior [22, 23]. Furthermore, also when monitoring benign programs or full systems, it is a desired goal to minimize the artifacts posed by the monitoring facility and force the monitored system to behave in a natural way as if it was running without any instrumentation (e.g., there is no difference in the order of scheduled tasks). By realizing the code interception from the *outside* of a system, maximum transparency can be reached which makes it hard for the malware to detect the analysis environment [7, 8].

In general, a monitoring facility is not able to hide itself from an object of surveillance that has the same or higher privileges, i.e., a kernel mode monitor can hide from a user mode malware, but cannot hide from kernel malware. Hence, by moving the monitor into the hypervisor, and thus operating on a higher privilege than the kernel itself, it is possible to even hide from malicious *kernel* code and analyze it in a precise way. Furthermore, current operating systems perform integrity checks that disallow the modification of kernel components. More specifically, the 64-bit version of Windows Server 2003 first introduced *PatchGuard* that implements this kind of defense mechanism. Thus, it is not possible to modify the system itself for monitoring from the inside on such platforms.

OS Independence One main advantage of our memory-based analysis approach is the fact that it works completely independent of the operating system. As long as the system is based on virtual page-based memory and the executed code is organized into different memory modules, our method can be applied. Unsurprisingly, this holds for all modern operating systems like Windows, Linux, and Mac OS X. However, while our monitoring approach is completely OS agnostic, the actual analysis component requires OS-dependent VMI routines to extract valuable data (i.e., we also need to address the *semantic gap* [1, 2]). For instance, we need to identify the called system functions and their parameter values to generate useful analysis results. Since most of the underlying concepts are implemented in the same or a very similar way in all commodity operating systems, the VMI techniques developed by us can be easily adopted to other platforms such as Linux or Mac OS X as well. Furthermore, we can leverage existing techniques to overcome the semantic gap problem [3, 4, 14].

Soundness Obviously, one inevitable requirement of an analysis system is the soundness of the generated results [24]. At first glance, monitoring solely on a page granularity (which is the only possible way when using TDP mechanisms as we will explain in the next section) offers only a coarse view of the executed code. However, it has turned out that this approach is practical since it delivers a sufficient amount of information and offers an efficient trade-off between performance and precision. While systems that operate on *instruction-* [9,10] or *branch-level* [25] granularity deliver much more details, they suffer a huge degradation of execution performance. Furthermore, for most applications this detail level lacks abstraction and requires complex data reduction methods to finally obtain the desired information. As a possible remedy, decreasing the granularity to the *system call level* may reduce the data to the necessary amount. Nevertheless, on this level relevant information may be missed, e.g., if certain operations are handled in user mode only and do not lead to a system call at all. Furthermore, many complex operations issued in user mode — as for example creating a new process — involve the execution of dozens or even hundreds of low-level system calls. Hence, monitoring on this level requires the aggregation of many recorded low-level syscalls to reconstruct the originally executed high-level operation. Adhering to these limitations and implications, we found that monitoring transitions between certain sets of memory regions fits best in practice. On that level, each high- and low-level library or system call can be precisely intercepted and monitored.

Flexibility Another design goal of our system is the flexibility to use it in multiple different scenarios. For example, it should be possible to analyze only selected processes, user-/kernel mode modules, or the complete system. To reduce the amount of collected data to a reasonable amount, it is thus necessary to choose between different granularities and scopes. In one case, only the transitions between the main module of an analyzed software and the rest of the system should be logged. In another case, the transfer between each single memory page of the monitored process should be considered.

Isolation Isolation is necessary when analyzing malicious code. In general it is inacceptable to let the malware disturb or otherwise influence the monitoring system. If that happens, the generated analysis results may get corrupted and, hence, can no longer be trusted. More critical is an evasion that temporarily utilizes the monitoring system to cause harm to other machines or even manages to permanently compromise the analysis environment. By assuming that there are no design or implementation flaws in the underlying hypervisor, we can be sure that there is no way for code running inside a virtual machine to affect the hypervisor and we can provide reasonable isolation.

Timeliness Existing malware analysis frameworks typically only operate on 32-bit systems, while the majority of machines in practice are nowadays powered by 64-bit processors and operating systems. Furthermore, the analysis environment is usually Windows XP SP2, a system that was released back in 2004. Both aspects indicate that current analysis frameworks are behind state-of-the-art systems and thus we need tools also capable of analyzing current malicious software such as 64-bit rootkits for Windows 7, something not possible with current tools.

1.3 Approach

In this paper, we propose and implement a method to utilize hardware features introduced by the latest generation of CPUs. We utilize these features for efficient system monitoring based on controlling the memory pages that are currently executable. More precisely, our approach consists of two phases. In a first step, we need to define those parts of the memory that should be monitored for executable code, a concept we call *Currently eXecutable Pages* (CXPs). During runtime, we utilize the hardware support of TDP to efficiently monitor how module boundaries are crossed: we maintain and switch between different TDP clones that partition the memory with respect to its executableness. This approach enables us to observe the behavior of a program or even a complete system since we can precisely track which memory region is currently active. Beyond analysis of malware, such a system can also be used for fine-grained profiling, for example to identify the regions of a program (or full system) that are frequently executed.

More specifically, the motivation behind our approach is to assist system analysts from several different application fields. Our main intended usage scenario is the analysis of malicious software. In this case, the most important requirement for an analysis system is to hide and protect itself from the analyzed piece of (malicious) code. Another ambitious demand is to extend the target scope of traditional analysis frameworks from simple user mode processes to kernel mode malware and especially 64-bit code, an emerging area that has received almost no attention up to now. Especially the latter task poses a serious problem on current Windows systems since such systems completely forbid the modification of kernel components for monitoring purposes and none of the existing malware analysis frameworks [9, 10, 15, 16, 24] supports 64-bit code so far.

A second application for our approach is transparent and efficient performance profiling of single applications or even complete systems without the need to modify the binaries or configuration of the executed components. By closely analyzing which code is currently executed we can assist profiling tools and for example spot hot memory regions (i.e., code executed frequently).

1.4 Contributions

In summary, we make the following contributions:

- We propose a new method to monitor code execution based on the concept of *Currently eXecutable Pages* (CXP), i.e., we closely track which memory regions are currently active and use this information as a basis for our analysis. We demonstrate how the *Two-Dimensional Paging* (TDP) mechanism available on current x86/x64 architectures can be used to analyze a full system (or only certain processes/memory regions).
- Beyond applications in the area of malware analysis, we show how this method can be used to perform system profiling (e.g., how much time is spent in which modules/memory pages) for performance optimizations (e.g., memory deduplication for VMs).
- We present CXPInspector, a prototype implementation of the proposed methodology. To the best of our knowledge, our tool is the first malware analysis environment capable of analyzing both user and kernel mode malware for machines powered by a 64-bit processor. The current implementation is tailored for Windows 7, but the general concept behind our approach can be applied to other operating systems as well since the memory analysis method is OS agnostic. We show that our prototype is capable of analyzing even sophisticated 64-bit kernel malware by monitoring a variant of the infamous TDSS/TDL4 rootkit. Furthermore, we show that our tool can also be used to generate detailed performance traces using the example of the Apache web server.

2 Background

To understand the design and implementation of our approach, we need to introduce some technical aspects first.

2.1 Memory Regions

Current operating systems (and applications running on top of them) typically organize their executable code into several files that are loaded as separate memory modules. In privileged mode, the kernel usually is accompanied by several loadable modules, no matter if a monolithic or a microkernel is used. In user space, the operating system offers dedicated runtime libraries to access the system resources (i.e., *Dynamic Link Libraries* (DLL) under Windows, *Shared Objects* (so) under Linux, and *Dynamic shared Libraries* (dylib) under Mac OS X).

Furthermore, most applications are divided into one executable binary plus several custom libraries, which continuously call functions from each other. For example, consider an application that invokes a function from one of its libraries. This function then wants to interact with the operating system and, hence, calls into a API function that is offered by some OS user mode library. In turn, this library invokes a correlating kernel function by performing the proper system call. The kernel delegates control to the corresponding device driver, again by calling into some driver function.

All of these executable (user and kernel space) files are mapped into the virtual memory space as separate and contiguous regions. Beside those file-mapped regions, a running system also maintains several dynamically allocated memory chunks, like thread stacks or heaps — both in user and kernel space. In the following, we refer to all of these adjacent memory ranges simply as *memory regions* and will no further differentiate between them.

2.2 Virtualization on x86/x64

The x86 platform has been a notoriously tough platform for virtualization due to the complex instruction set architecture and historic lack of hardware support for this purpose. The first generation of virtualizers therefore employed a variety of tweaks and tricks in order to implement efficient and secure virtualization of guest VMs. This usually involved complex code rewriting techniques and further required to emulate guest code in certain situations. In order to facilitate and ease the development of virtualization solutions, both Intel (Intel VT [26]) and AMD (AMD-V [27]) have introduced hardware support for virtualization in the past. In a nutshell, both features provide an additional privilege level (referred to as *ring -1*) along with additional instructions for setting up and controlling VM guests.

Intel VT provides a *virtual-machine control structure* (VMCS) for each virtualized CPU that describes various settings of the guest VM (e.g., the virtualized address space, register values, hooked interrupts, etc.). This VMCS can be read or modified using the dedicated machine instructions `vmread` and `vmwrite`. If the hypervisor wants to continue execution in guest mode, it executes a `vmenter` instruction that switches the CPU into ring 0 or ring 3. The CPU then proceeds to execute the guest code normally until an event occurs that must be handled in the hypervisor (e.g., device I/O). This triggers a so-called *VMExit* that causes a transition to ring -1 in which the hypervisor may then handle the exception and eventually continue guest execution by issuing `vmenter` again. The concrete events that lead to a *VMExit* can be determined by reading the appropriate flags in the VMCS. The mechanism is similar for AMD-V.

2.3 Two-Dimensional Paging

One important aspect of our proposed approach is the hardware support for separating physical memory of a guest system. The virtualization layer usually allocates a set of physical frames for each specific guest and each guest may only access this specific set. Since the MMU itself was not virtualized in both AMD-V and Intel VT in the beginning, hypervisors had to emulate the

MMU by using so called *shadow page tables* (SPT). When SPT are in use, the page tables that the guest VM works with are actually never used in the hardware. Instead, the hypervisor uses a modified copy of these page tables (i.e., SPTs) and intercepts all VM operations related to the paging mechanism. This happens transparently to the guest's memory management system.

The drawback of this approach is that it requires significant implementation effort on the hypervisor side. Furthermore, it introduces a non-negligible performance overhead because every page fault that happens in the guest at first has to be rerouted to the hypervisor. This is necessary since certain page faults must be handled by the hypervisor (such as when the guest modifies page tables). Additionally, all direct accesses to the guest paging tables have to be intercepted to propagate the changes to the corresponding SPT entries. Finally, also the load and store operations of the guest CR3 register have to be monitored and appropriately emulated by the hypervisor to make the shadowing mechanism completely transparent to the guest.

To overcome these weaknesses, MMU virtualization in the form of *Two-Dimensional Paging* (TDP) was introduced recently. It basically adds another step at the end of the MMU translation, i.e., from a guest physical address to a host physical address. Therefore, the physical guest memory is virtualized and can be mapped to arbitrary host memory frames by the hypervisor. This guarantees address space separation and it is no longer necessary to hook the page fault handler. To that end, TDP requires the hypervisor to specify a completely new set of translation structures for the guest-physical to host-physical translation. These structures are built in the same way as the normal paging structures, i.e., on 64bit there is a 4-level paging mechanism. Although Intel and AMD have invented their own implementations of TDP, they only differ in small details. While Intel calls their system *Extended Page Tables* (EPT), AMD refers to it as *Nested Page Tables* (NPT). For the sake of brevity, we focus on the details of Intel's EPT implementation in our work. Appendix A presents more details on this mechanism and clarifies the TDP address translation with further examples.

3 System Overview

Based on the background information provided in the last section, we can now describe the general approach and technical details of our implementation.

3.1 General Approach

The basic idea of our approach is to dynamically partition the main memory of a virtual machine into two distinct parts, from which one is executable and the other non-executable. By adjusting the position and size of the *Currently eXecutable Pages (CXP)* dynamically, we are able to intercept control flow transitions between different entities of the running system. Consequently, different CXP arrangements result in different tracking scopes and granularities. Our current system supports the following three CXP scopes:

(a) one memory region

(b) a set of memory regions

(c) one single memory page

In this context, we refer to one memory region as one contiguous virtual memory block (i.e., program executable, custom or system library, kernel, kernel driver, stack, heap, and so on). The first two scope types are illustrated in Figure 1. In case (a) each memory region is handled separately and the CXP always corresponds to that particular region which is currently executed, i.e. which is pointed to by the instruction pointer. This enables the detection of *all* intermodular function calls and their corresponding returns, no matter if they cross the user space/kernel space barrier or not. The second case (b) allows to build sets of certain modules, e.g. one application set A (program executable and its custom libraries) and one system set B (user mode system

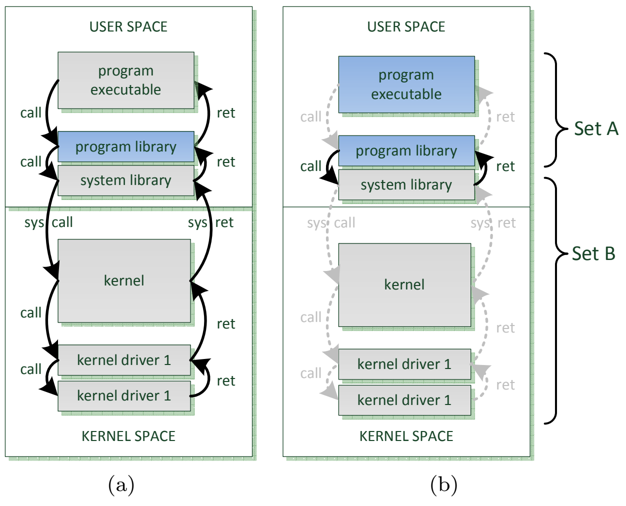

Figure 1: CXP spans (*a*) one or (*b*) a set of memory regions

libraries, kernel, and kernel drivers). By switching the CXP between these sets, only the calls between an application and the operating system are monitored and the calls between different components of the application or the system are ignored. When using the last partitioning type (*c*), each set CXP spans only one virtual memory page. Hence, it offers the finest granularity and enables us to detect all code flow between separate memory pages, no matter if they belong to the same memory module or not. Obviously, it has the largest performance impact, since we have to intercept running programs very often.

No matter what scope is used, each time a control flow to an address outside the CXP takes place, a *VMExit* is triggered and the hypervisor becomes active. Its task then is to log information about the code flow and mark the newly reached memory region as executable while marking the last CXP non-executable. Depending on the used scope, different information may be logged in such events. For the cases (*a*) and (*b*), interception always happens on intermodular (sys-)calls or returns. We reconstruct the name of the called function and the parameter values that were used. Optionally, we also record the memory addresses from which the call/return was originating. For that purpose, we utilize the branch recording facility of modern Intel/AMD [26, 27] processors. When enabling this feature, source and target addresses of the last performed branches are stored in certain *Model Specific Registers (MSR)* of the processor. Since we use case (*c*) only for profiling measures, the relevant information in that case is the base or entry address of the different visited memory page.

In general, our approach is strongly based on hardware features and can be used independently from the virtualized operating system. Nevertheless, there are certain situations that require a semantic understanding of certain memory contents and, hence, need to be customized for the particular used OS to enable virtual machine introspection [1].

3.2 Prototype Implementation

We have implemented a prototype of the proposed approach in a tool called CXPInspector. The current implementation supports the 64-bit version of Windows 7 as virtualized environment, but it can easily be adopted to the 32-bit version and with slightly more effort also other operating systems. CXPInspector itself is based on the hardware virtualizer *KVM* [28] and, thus, has to be executed on a Linux host platform. KVM is a kernel driver that implements a hypervisor to provide hardware-based virtualization using either Intel VT or AMD-V. The driver provides an interface to the QEMU toolset which is responsible for handling all emulation tasks such as

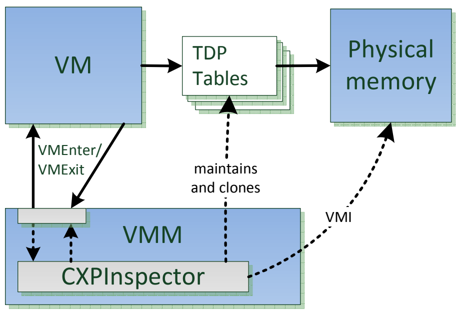

Figure 2: Schematic system overview

providing a network and graphics card. The advantages of using KVM over similar hypervisors are that it is open source, the code is reasonably small, it supports both AMD and Intel systems, and the plug-in load/unload architecture of linux drivers eases the development. Moreover, KVM supports Two-Dimensional Paging as discussed in Section 2.

As shown in Figure 2, our tool extends the hypervisor and interacts with the virtualized components in several ways. First of all, it controls the TDP tables that are used to map the virtualized guest memory onto the real physical host memory. While in a regular virtual machine only one set of TDP tables is used, our system maintains several slightly modified *clones* to switch between different CXPs. By specifically crafting these clones, the code flow between different memory regions can be monitored and intercepted. For example, if the target location of a jump instruction is marked non-executable in the currently used TDP clone, a *VMExit* will be triggered when the jump is executed. By inspecting the current virtual CPU state and reading from the virtualized memory (i.e., performing virtual machine introspection), it is then possible to gather detailed information about the currently executed instructions.

In the remainder of this section, we provide more technical details on how memory regions are managed internally, how the TDP clones are created, and how our introspection features work.

3.3 Monitoring Memory Regions

The main purpose of CXPInspector is to catch the control flow transitions from one module to another in order to inspect library and system calls. Since we need to partition the *complete* virtual memory, we have to extend the unit of observation from modules to the more general concept of *memory regions*. As explained in the beginning of this section, a memory region in general is a chunk of virtual memory that has been allocated as one single contiguous block. Accordingly, besides executable modules the set of memory regions comprises all dynamically allocated memory structures like heaps, stacks, system control blocks, or interrupt service routine stubs.

During runtime our system keeps track of all existing memory regions and maintains information about them within different lists. When a certain memory region is accessed for the first time, CXPInspector checks if the hosting memory belongs to a process' user mode address space or if it is located within shared kernel space. In the first case, the memory region is stored into a process-related list and otherwise it is maintained in the global kernel space region list. If a memory region is no longer used, it is removed from its hosting list and, if a process is terminated, all associated user space regions are freed as well.

3.4 TDP Table Cloning

To implement CXPs efficiently, we provide specially crafted clones of the actual TDP tables for each memory region in a monitored process. In each clone we set all page frames associated with the corresponding region to *executable* and mark all other frames as *non-executable*.

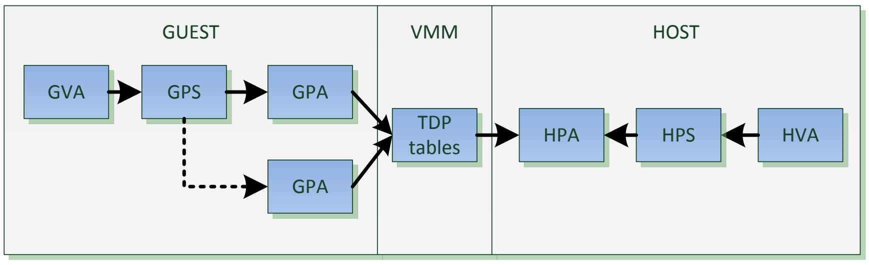

Figure 3: Overview of address translation process

One crucial point is that the currently used TDP clone is always up to date to work properly and avoid missing a memory region transition. Since the TDP structures are constantly subject to modifications by the hypervisor, it is mandatory to synchronize the changes between the existing clones. Such modifications may have several causes. First of all, TDP entries are created *lazily on-demand*, i.e., a frame is not allocated in the TDP unless it is used for the first time. Besides that, the host operating system may opt to swap out frames which are currently in use by a guest VM. In that case, the TDP entries of the affected frames have to be adjusted appropriately. Finally, also the guest operating system may invalidate and re-validate frames when the associated memory gets paged-out or paged-in. As an effect, the guest physical frames that host a certain memory region changes. Since utilizing TDPs to control execution can only be based on physical guest addresses — and not on virtual ones — we have to react on those guest induced modifications as well and update our TDP clones accordingly.

After all, for maintaining consistent TDP clones and performing valid VMI operations, our tool has to consider many different addressing layers. On the one hand, we have to consider the guest's virtual address space and, on the other hand, we work with TDP tables that contain physical addresses, both from the guest and the host. Hence, we need to translate between *guest virtual addresses* (GVA) and *guest physical addresses* (GPA). As illustrated in Figure 3, this is achieved by a page walk on the *guest paging structures* (GPS). In order to locate the guest memory regions in host physical memory, their GPAs have to be translated into *host physical addresses* (HPA), which is done by using the TDP tables. Note that also the GPS themselves are referenced via GPA and, hence, their addresses have to be translated to HPAs during the page walk (this is indicated by the dotted line in the figure). In order to actually read the memory contents from the host (where our tool is located), we need to translate these HPAs into correlating *host virtual addresses* (HVA). To that end, we utilize several data structures that are maintained for that purpose by the KVM driver.

As explained before, all of these mappings may constantly change while a process is running, e.g., if the guest or host operating system swaps out a page or reloads it from the page file using a different frame. Monitoring the host side is easy since the Linux operating system and the KVM driver already offer appropriate callback functions for that purpose. So called *MMU notifiers* inform the loaded kernel modules whenever the HPS are modified. Propagating the TDP updates to our clones is also not a problem since all modifications are done inside the KVM driver and we can modify the relevant code. However, when using the TDP feature, there is no KVM support for monitoring GPS modifications and, thus, we have to do it on our own. To that end, the guest page tables are traversed starting from the current guest `CR3` register. CXPInspector sets the TDP entries of all frames that hold GPSs to *non-writable*. Therefore, if the guest operating system modifies the address space of a monitored region, a TDP violation is raised and handled by the hypervisor. In that event, we read the GPS entry which is to be overwritten, temporarily mark the related memory as writable, and re-execute one instruction using the *monitored trap flag* (MTF) hardware feature. Since we use MTF, we do not have to emulate the instruction that causes the EPT violation. Lastly, we set the memory to non-writable again.

3.5 Introspection

To obtain actual and detailed insights into the behavior of a program (or the complete system), CXPInspector must obtain some information about the guest's operating system data structures. For example, a list of all running processes is required to observe which programs are active. In addition, all loaded regions within a monitored process and all kernel driver regions have to be known by the hypervisor. Furthermore, to generate valuable analysis results, not only the memory addresses of control flow transitions should be logged, but also information about the called function and the used parameter values should be given.

Our system makes use of a variety of known techniques in the field of VM introspection [1, 3, 29, 30]. Note that our system is tailored to Windows 7 and to the best of our knowledge CXPInspector is the first analysis system that supports this operating system in 64-bit mode. Supporting other operating systems is easily feasible since only the introspection component needs to be adjusted, all other components of CXPInspector are OS agnostic.

The first step of introspection is to determine the base address of the loaded kernel. Since current operating systems employ *Address Space Layout Randomization (ASLR)* to generate different base addresses each time the system is booted, we need to obtain it dynamically while the system is running. To that end, we monitor the `GS_BASE` MSR that is written on 64-bit systems during thread switches. When this MSR is loaded with a meaningful value for the first time (i.e., when the first system thread is created by the OS), we know that the current instruction pointer points to some code within the loaded kernel. By starting from this code address and searching the memory in a backwards direction for a valid module start signature, we are enabled to determine the kernel start address.

In addition to this we use the publicly available database of Microsoft kernel symbols to locate certain important symbols inside the kernel image, such as the head pointer of the process list. In case such debugging symbols are not available for the given kernel, we fall back to parsing the PE exports of the kernel or we can use automated techniques to overcome the semantic gap [3, 4]. Additional information then can be gathered by traversing the corresponding kernel data structures, such as the loaded kernel modules or the list of all running processes. For each process the list of loaded user space modules can be obtained by traversing its *virtual address descriptors* (VAD). This list holds all allocated memory regions of the user space for the given process.

Most data structures accessed during introspection are managed by the kernel and thus are located in kernel space. This has the advantage that they usually cannot be paged-out by the operating system. Whenever CXPInspector parses such structures, it can be sure that they are indeed present in the guest's memory. However, this does not hold for user space memory, since this kind of memory can be swapped out. For example, when a string pointer is passed to a string copy function, the string itself might not be present in memory at the time of the call. CXPInspector handles this situation by first injecting a page fault into the guest system to force the guest's page fault handler to page-in the required memory.

The situation is a bit more complicated for pageable kernel space memory. Though parts of the kernel space generally cannot be paged out, this does not hold for the entire kernel memory. For example, the operating system might choose to swap out parts of a driver image after it has fully initialized. To access these memory regions, we also utilize the injection of page faults. However, we have to delay them until the guest system is running in ring 0 and the current IRQ level is set to the lowest value. Otherwise, an application fault or kernel panic could occur if we inject the page fault immediately.

To enhance the generated analysis logs, CXPInspector enriches them with detailed information on the called functions and their argument values. To retrieve the name of a called function, we either use symbol information (if available) or parse the PE header of the module. Additionally, we parse the header files of the Windows SDK to obtain the number and types of function parameters and the names of public Windows API functions. Our tool parses the corresponding C header files with the help of *Doxygen* and extracts the prototypes of exported functions which are part of the operating system API. When a module transition occurs, CXPInspector inspects the stack of the guest and reads the arguments of the called function. If parameter type information is available, metadata is also parsed and logged. For example, if we know a certain parameter is a string pointer, then the string data is read from the guest's memory and saved in the log.

3.6 Enforcing Transparency

One of our main goals is transparency towards the monitored system. While there exist dozens of — simple to sophisticated — methods to detect current analysis frameworks [22, 23], utilizing modern hypervisor-assisted virtual machines renders most of them useless. Nevertheless, timing attacks are a simple but effective detection scheme that is applicable to detect hypervisors. The actual timing can be performed with the help of external or internal timing sources. Using external sources is complex to implement, but nearly impossible to avoid by the monitoring system. Attacks using internal timers utilize the processor's *Time Stamp Counter* (TSC) to measure the needed time for certain operations and differ between native and instrumented systems. Fortunately, modern hypervisors offer an easy approach to manipulate the TSC to conceal their own computation time from the virtual machine. To that end, a customizable *Time Stamp Counter Offset* [26] can be specified that is added/subtracted to the real TSC value each time it is queried by a guest. Our tool simply measures and cumulates the time that is spent outside the virtual machine and sets the TSC offset accordingly.

Optionally, we utilize the processor's *Branch Tracing* (BT) [26] facility to obtain the originating memory address for each branch/return instruction. Since the MSRs related to BT are only accessible from privileged code, user mode guest processes are not able to detect this. Nevertheless, kernel mode code could interfere with our utilization or derive the information that it gets monitored. Therefore, we configure the virtual machines in a way that all accesses to the related MSRs are intercepted and simulate the corresponding instructions in a way that hides our utilization from the guest.

4 Evaluation

In order to demonstrate the practicality of CXPInspector, we show how the tool can be used to analyze some of the most advanced malware currently known in the wild. Furthermore, we measured the performance overhead incurred by our current prototype and discuss how CXPInspector can be utilized to perform system profiling (e.g., how much time is spent in which modules/memory pages).

4.1 Malware Analysis: Case Study on TDSS/TDL4

To show that our system is capable of providing deep insight into the behavior of a sophisticated malicious piece of code, we analyzed a recent variant (called *Purple Haze*) of the well-known 64-bit rootkit TDSS/TDL4 [31, 32]. To the best of our knowledge, no existing automated malware analysis system is capable of analyzing such 64-bit rootkits. For this test we have configured CXPInspector to monitor each region separately (case (a) in Section 3.1).

Overview of TDL4 The TDL4 rootkit family is one of the few sophisticated malware samples that are capable to infect 64-bit Windows kernels. It modifies the system's boot loader and disables the Windows *PatchGuard* protection at boot time. Without PatchGuard being enabled, the rootkit can then tamper with kernel code and data that would otherwise be protected. TDL4 also creates a hidden disk partition with a custom filesystem where it stores all of its binaries along with some configuration files. After disabling PatchGuard, the malicious boot loader loads a custom rootkit driver from this hidden disk partition. Obviously, the driver itself is not registered as a regular kernel driver; it is manually mapped into a free region of the kernel space instead. TDL4 installs a variety of hooks in the kernel by overwriting crucial kernel functions and pointers. For example, it intercepts kernel requests to the `atapi.sys` driver that is responsible for instructing

```
from=fffff8000262f8fb  to=fffff800026a9cd5   ntoskrnl.exe:KiApcInterrupt+c5
from=fffff800026a9e01  to=fffff880010146b6   ataport.sys:+a6b6
from=fffff880010114f9  to=fffffa8002949229   <unregistered_region>:+229
from=fffffa8002949242  to=fffff800026d6010   ntoskrnl.exe:KeWaitForSingleObject+0
from=fffff800026d61ca  to=fffffa8002949248   <unregistered_region>:+248
from=fffffa8002949272  to=fffffa800294b402   <unregistered_region>:+402
from=fffffa800294b78d  to=fffff8000265d1ec   ntoskrnl.exe:_strnicmp+0
                                                _strnicmp(_Str1="ph.dll", _Str2="phdata")
from=fffff8000265d237  to=fffffa800294b793   <unregistered_region>:+793
                                                _strnicmp returns 0xffffffca (-54)
from=fffffa800294b78d  to=fffff8000265d1ec   ntoskrnl.exe:_strnicmp+0
                                                _strnicmp(_Str1="phx.dll", _Str2="phdata")
from=fffff8000265d237  to=fffffa800294b793   <unregistered_region>:+793
                                                _strnicmp returns 0x00000014 (20)
from=fffffa800294b78d  to=fffff8000265d1ec   ntoskrnl.exe:_strnicmp+0
                                                _strnicmp(_Str1="phd", _Str2="phdata")
from=fffff8000265d237  to=fffffa800294b793   <unregistered_region>:+793
                                                _strnicmp returns 0xffffff9f (-97)
from=fffffa800294b701  to=fffffa800294c0c6   <unregistered_region>:+c6
from=fffffa800294c67d  to=fffff800026e2adc   ntoskrnl.exe:ExReleaseFastMutexUnsafe+0
```

Figure 4: Trace of a disk write request in presence of TDL4 hooks

the disk controller to read or write data blocks. By doing so, it manages to hide its self-crafted partition on the system's disk. Moreover, it monitors process creation to infect new system processes. Hence, whenever a new process is created, it injects a user mode component into the new address space.

We executed the TDL4 sample within CXPInspector and recorded a trace of the executed kernel code. In the following, we explain the results obtained with the help of our tool and some excerpts are shown in Figure 4 and 5.

Disk Controller Hooks Whenever a request to read or write data to or from a disk is issued, the operating system creates a so-called *I/O Request Packet* (IRP). The IRP is then subsequently processed by all the different drivers that are associated with the used disk device. This (among others) includes the filesystem driver, the partition management driver, and eventually the disk controller driver. TDL4 overwrites the code of the atapi IDE driver in the last stage with a jump instruction to its own code. Thus, it makes sure that it does not miss any disk read or write requests. The task of the hook itself is to make sure that the hidden rootkit partition remains invisible to the remaining system. Figure 4 shows an excerpt of a logfile created by CXPInspector during a disk IRP call. Since the TDL4 rootkit driver is not registered in the system's driver list, jumps to its code region are tagged as <`unregistered_region`>. The log shows that the atapi driver code jumps into the rootkit, where checks are performed to see whether the request targets a file on the hidden partition (note that all files in that partition start with "`ph`").

Process Creation Hooks In order to detect newly created processes, TDL4 hooks into the process creation code of the kernel by using the *Event Tracing for Windows* (ETW) kernel API. Whenever a context swap occurs, a notification callback routine in the rootkit is called. Figure 5 presents another excerpt from the log file generated by CXPInspector, more specifically the figure shows how TDL4 acquires a handle to the newly created process by calling `ZwOpenProcess`. The malware then builds a string to its configuration file (`phdata`) and reads the contents. In the end, TDL4 proceeds to inject its user mode payload into the new process.

The log file generated by CXPInspector shows that our system can precisely monitor the execution traces of a sophisticated rootkit such as TDL4. They paint a clear picture of which kernel modules are tampered with and by which techniques TDL4 manages to hide its presence.

4.2 Spotting Hot Memory Regions

Besides malware analysis, another application of our system is transparent and efficient performance profiling. CXPInspector allows the profiling of selected modules, processes, or the com-

```
from=fffff800026216e9  to=fffff800027d51d2   ntoskrnl.exe:EtwTraceContextSwap+92
from=fffff800026da15f  to=fffffa8002949edc   <unregistered_region>::+edc
from=fffffa8002949f36  to=fffff800026c5920   ntoskrnl.exe:ZwOpenProcess+0
                                               ZwOpenProcess(ProcessHandle=0xfffff88002fabcb0,
                                                   DesiredAccess=0x1fffff,
                                                   ObjectAttributes=0xfffff88002fabc70,
                                                   ClientId=0xfffff88002fabc60)
[...]
from=fffff800026da352  to=fffffa800294c700   <unregistered_region>:+700
                                                  RtlInitUnicodeString returns <void>
from=fffffa800294c75e  to=fffff800026c5f00   ntoskrnl.exe:ZwCreateFile+0
                                               ZwCreateFile(FileHandle=0xfffff88002fabc58,
                                                   DesiredAccess=0x100003,
                                                   ObjectAttributes=0xfffff88002fab928,
                                                   IoStatusBlock=0xfffff88002fab8f0,
                                                   AllocationSize=0x0,
                                                   FileAttributes=0x00000000 (0),
                                                   ShareAccess=0x00000001 (1),
                                                   CreateDisposition=0x00000003 (3),
                                                   CreateOptions=0x00000022 (34),
                                                   EaBuffer=0x0,
                                                   EaLength=0x00000000 (0))
from=fffffa800294c481  to=fffff8000271a740   ntoskrnl.exe:_snprintf+0
                                               _snprintf(str=0xfffff88002fab050,
                                                   size=0x00000103 (259),
                                                   format="%.*S")
from=fffff8000271a7e4  to=fffffa800294c487   <unregistered_region>:+487
                                               _snprintf returns 0x0000002d (45)
                                      (str="{752492e1-ed26-91d2-750b-04be2c7925eb}\\phdata")
```

Figure 5: Trace of a process creation call in the kernel

plete system, while offering this for user and kernel mode on 64-bit systems. When monitoring with page granularity (refer to case (*c*) in Section 3.1), our tool measures the execution time that is spent in each single virtual memory page. These values can be examined to spot the hot regions of a running system.

Though the minimum granularity of this measurement is a memory page, the gained profiling results can be reasonably used in many different ways. For instance, they can be used by developers to find bottlenecks and other optimizable portions in their code or they can assist the operating system to select a better swapping strategy. Additionally, they can be used to improve the effectiveness of *Kernel Samepage Merging (KSM)* [33]. KSM is an optimization feature of KVM that shares duplicate pages between different virtual machines on a physical memory level. The identification of such similar (and hence duplicatable) pages requires hash lists and the more pages are shared, the longer the list lookup takes. Therefore, it is possible to speedup the whole process by limiting the amount of shared pages. This requires an optimal selection of candidate pages and these can be obtained by our performance monitoring. In a similar use case, such information could also be used to improve memory deduplication in the context of virtual machines [34, 35].

To illustrate this feature, we have profiled an *Apache* web server on a 64-bit Windows 7 system. We used a virtual machine with 1 VCPU and 2 GB RAM running on an Intel i7-2600 CPU with 3.4 Ghz and 8 GB of physical memory. For measuring we have applied the performance testing tool *ab* that is distributed with the Apache server, and scripted it to access 1,000 times a copy of our chair's webpage (11kb size). It should be noted that the collected performance values do not only reflect the 1,000 page hits, but also the initialization phase of the web server itself.

Figure 6 provides an overview of the aggregated execution time per memory module. The x-axis shows the module names that consumed the most execution time during our tests. The y-axis presents the aggregated number of CPU cycles that were spent in all pages that belonged to each module. As one can see, the two most occurring regions are the Windows kernel *ntoskrnl.exe* (approx. 120 billion ticks) and the Native API *ntdll.dll* (35 billion ticks). This is not surprising, since these two modules constitute most parts of the operating system's core functionality. For better visibility, the figure only contains the top most visited 20 regions, while the test delivered 127 different memory regions in total:

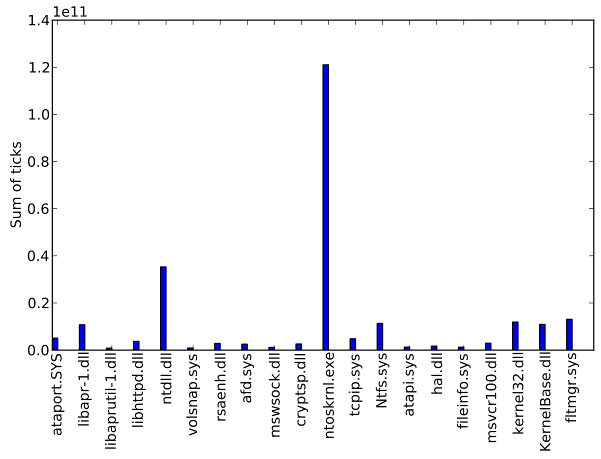

Figure 6: Performance profile of the top 20 modules

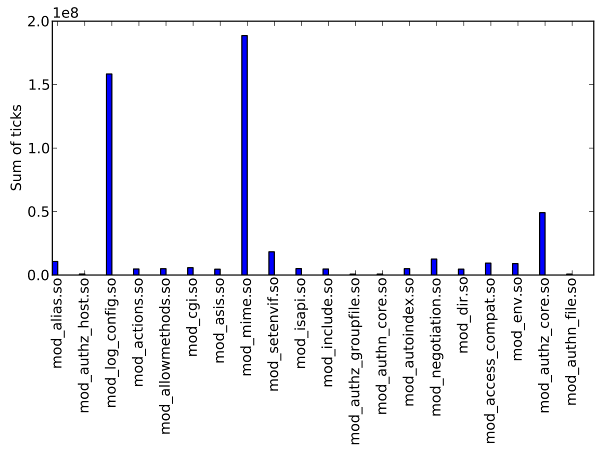

Figure 7: Performance profile of the top 20 Apache modules

- 62 user mode modules, containing
  - the Apache image (bin/httpd.exe)
  - 22 Apache modules (modules/mod_*.so)
  - 5 Apache libraries (bin/lib*.dll)
  - 1 dynamic memory region
  - 33 Windows DLLs
- 65 kernel mode modules

As a second example we analyzed the collected profiling data of the 20 top most used Apache modules. Figure 7 shows that most time is spent in *mod_mime.so* and *mod_log_config.so*.

Finally, we have evaluated the per page profiling values of one particular module. To that end, we have chosen one of Apache's core components, namely *libhttpd.dll*, simply because it's corresponding profiling data is well suited to illustrate the characteristic value distribution of a

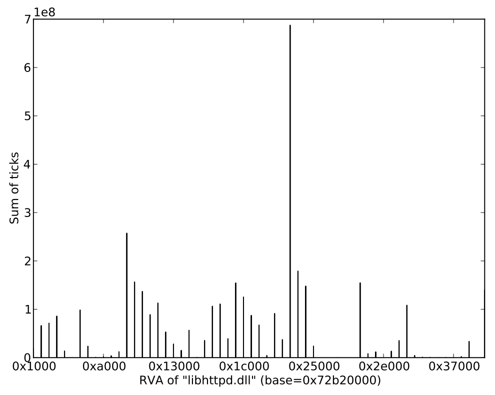

Figure 8: Performance profile of libhttpd.dll

	EPT	SPT	Case (a)	Case (b)	Case (c)
Mean value (ms)	1.271	1.485	51.021	3.117	80.172
Factor	1.00	1.17	40.16	2.45	63.10
Min value (ms)	0.734	1.107	50.151	2.815	78.908
Factor	1.00	1.51	68.33	3.84	107.50

Table 1: Minimum and mean values

certain module. Figure 8 shows a high peak in the page ranging from RVA 0x22000 to 0x22fff. We have inspected the binary with the help of a disassembler and found out that this memory region contains a bunch of functions that are associated with string operations and regular expressions. This seems very plausible, because for the used scenario most work of the web server lies in pattern matching and string comparisons.

A different possible application of this profiling feature is the fingerprinting of a running system from the hypervisor: by using the characteristic profiling value distribution of certain modules, processes, or even the complete operating systems, a service provider is able to identify the components that are active within a running virtual machine *without* actually performing virtual machine introspection. We leave the actual generation and verification of appropriate profiling signatures as future work, but are confident that it is possible based on the results obtained with our prototype.

4.3 Performance

As a final experiment we have tested the performance impact of our current prototype implementation. To obtain meaningful measurements, we have to use tests that actually trigger the sources of the performance overhead imposed by our system. To that end, we first had to point out which operations actually take more time when executed within our modified environment. Though we have extended some execution paths within different places of the KVM source code, the majority of additional computations is done in the *TDP fault handler*. Furthermore, when using our system, much more *TDP faults* are triggered, namely one for each transition into or out of the CXP. Hence, to obtain reasonable performance results we had to apply tests that invoke those region transitions. We tested several COTS benchmarks but unfortunately, they largely focus on CPU-heavy computations and not on the interfaces between the application and other libraries. Applying these benchmarks would significantly underestimate the runtime overhead. Since there is not much value in developing synthetic tests for that particular purpose, we have used the setting from the previous section to gather performance measurements.

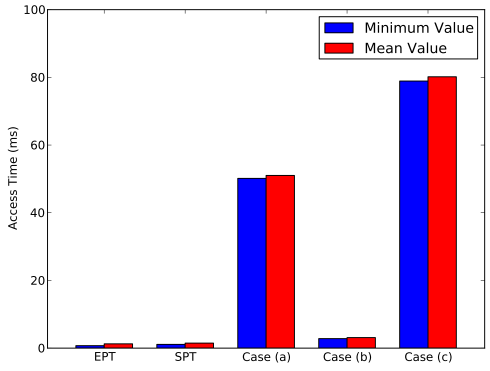

Figure 9: Performance overhead

More specifically, we benchmarked CXPInspector in the three different modes and executed the Apache benchmark tool *ab* to query the sample webpage for 10 * 1,000 times. The calculated minimum and mean values of these tests are depicted in Figure 9, while the concrete values are shown in Table 1. All numbers are compared to an unmodified KVM system (as described in the previous section) with EPT enabled (see first column). The second column shows the results from a slightly modified system, on which EPT was disabled and KVM hence forced to use shadow page tables instead. As can be seen, the effect of that is only a slight performance penalty. Columns three to five represent the cases (a) to (c) as described in Section 3.1 (i.e., (a) monitor each region, (b) monitor sets of regions, and (c) monitor each single page). For case (b) we have divided the complete memory into two region partitions: one that contained all the executables that belong to the Apache process and another with the remaining ones. The table and the graphic show that monitoring each region or even page separately imposes a significant performance penalty. However, building memory region partitions as done in case (b) results in only a neglectable performance overhead. The runtime factor for that case is 2.45, meaning a performance overhead of 145% which can be even further reduced as discussed in a later section. As the TDL4 example has shown this scope of monitoring is sufficient in the most cases for malware analysis. Hence, our results indicate that CXPInspector has in practice an overhead that is completely reasonable and much faster compared to emulator-based approaches [9, 10].

5 Related Work

In the past years, several dynamic analysis platforms were introduced that are geared towards analysis of malicious software. These tools work at different privilege levels and granularities. Many analyzers monitor the system calls of targeted modules to induce the runtime behavior. In user space, this is mostly done by either emulating the API [36], patching the import address table (IAT) [37], or by inserting trampoline jumps directly into the code of API functions to detour them into dedicated monitoring libraries [24, 38]. All of these approaches have in common that they modify the target's process address space in memory, either by overwriting code or by patching function pointers. A natural disadvantage of this approach is that it can be easily detected by the malicious code itself, which may then choose to cancel its execution or only exhibit benign behavior.

Another class of malware analyzers leverages whole system emulators (such as QEMU [39]) to run the to-be-analyzed program in a completely emulated environment and then monitors the API interface from the emulator [9, 10, 15–17]. This approach typically has a non neglectable performance overhead, but allows for a more fine-grained analysis and increased transparency, given that the emulator works correctly. Due to the extraordinary complexity of modern CPUs (especially x86-based ones), it is virtually impossible to implement an emulator without any imperfections and these subtle differences compared to native execution can again be used by malware to detect whether it is running in a emulated environment [40–42]. Moreover, some approaches use bare-metal, hardware-assisted CPU features in order to analyze code more transparently [25]. One problem with many hardware-based analyzers is that they only provide coarse-grained analysis reports since they do not incorporate advanced features such as TDP. For example, some analyzers only catch the interaction between user mode and kernel mode by hooking system call instructions [14]. However, some API calls are handled completely in user mode and thus cannot be seen by these tools. Furthermore, one API call from the target module's code may lead to plenty of other system calls as a side-effect. For example, a simple file copy API function call such as `CopyFile` under Windows leads to tons of file I/O calls on the system call interface since the code of `CopyFile` has to open/close files and perform several reads and writes. Consequently, analysis at the system call level inevitably loses abstraction. Even worse is the situation for those analyzers that use single-stepping to achieve more fine-granular analysis [8]. This further comes with a significant performance loss that can even be higher compared to emulator-based solutions. In contrast, our approach offers a better performance. Several hypervisor-based analysis approaches were proposed in the past [43–46], but none of them utilizes Two-Dimensional Paging. Since our approach is purely memory based and leverages low-level information, it is more generic in the sense that we can monitor different kinds of systems using our method.

Another relevant research area focuses on bridging the semantic gap that occurs in VM introspection [1, 3, 4, 29, 30]. Since the hypervisor only has a low-level, hardware-centric view in the guest, additional effort is necessary to transfer the semantic meaning of operating system data structures into the hypervisor. We take advantage of prior work in this area and leverage the proposed techniques.

Closely related to our work is IntroLib [47], a tool that traces user-level library calls with low overhead and high transparency. IntroLib is based on KVM [28] and uses hardware virtualization and the CPU's memory translation facilities to transparently mark certain memory regions in the guest VM as executable or non-executable. Whenever the instruction pointer is inside application code, the hypervisor sets the executable flags for all of the application's code, the rest of the memory is marked non-executable and vice versa. As a consequence, transitions between modules in these regions (e.g., API calls) can be caught and logged in the hypervisor. However, IntroLib has several shortcomings compared to our system: first, it only works with user space components under 32-bit Windows XP. This makes it impossible to monitor sophisticated malware that works on the privileged kernel level (such as rootkits). Second, IntroLib only separates the guest VM's address space in two *static* partitions (similar case (b) from Section 3.1): the to-be-analyzed code (its program image and its custom libraries) and the rest. It thus only monitors direct transitions from and to the analyzed modules and is incapable of seeing what happens beyond the first module in the entire chain of function calls. Third, IntroLib relies on the use of *shadow page tables*, which requires significant implementation effort in software to allow for guest address space isolation from the host. This also results in additional VMExits, which introduces an overall slowdown. In contrast, we use hardware-assisted TDP as discussed in Section 3.

V2E [48] is a system that also leverages nested page tables. However, the authors only use nested page tables for recording execution traces. The actual analysis then happens while replaying the recorded traces in an emulator.

6 Limitations

In the previous sections, we have presented CXPInspector, our approach to utilize hypervisors and hardware support for system monitoring. We now discuss the limitations of our approach and the current prototype.

The results of the performance benchmarks are twofold. A low latency is only reached by case (b), which divides the memory regions into two partitions and only monitors the transitions between them. This procedure is sufficient for many use cases, especially when monitoring malware. When monitoring transitions between *all* individual regions as in case (a), the observed latency is rather high. However, we are confident that this performance impact can be reduced tremendously by utilizing new CPU features available in the next generation of CPUs. At the moment, we invalidate the *complete* guest EPT caches once we switch or modify the contents of the TDP tables (i.e., each time a region transition occurs). The KVM source code and one Intel whitepaper [49] state that it is possible to invalidate only *certain* cache entries explicitly. Unfortunately, the hardware we used does not seem to support this feature (yet). It is obvious that invalidating only a few cache entries instead of discarding all address mappings would increase the performance of our tool dramatically. We are thus confident that next-generation CPUs with appropriate hardware support will lead to significantly better performance.

Another important aspect is the transparency of our system towards analyzed malware. While the only system modifications take place *outside* the virtual guest, there is no direct way to detect our tool. Nevertheless, there are several ways an attacker can notice the existence of a virtual machine in general, and there is further the possibility to indirectly discover our modifications by timing attacks. VM detection in general is an old problem [50, 51] and has been addresses in the past [22, 23, 52]. However, while many productive systems have moved to virtual machines by now, it is questionable if it is still a wise choice for malware to refrain operating in such environments. Our prototype has the same drawbacks as other dynamic analysis frameworks (e.g., conditional code obfuscation [53] and similar obfuscation techniques [54, 55]), but we can apply standard techniques to address such drawbacks (e.g., multi-path execution [19], identification of dormant functionality [56], and similar techniques [57, 58]). Furthermore, while we are utilizing hardware-assisted hypervisors, it is also easier for us to hide from detection. Such hardware can be configured in various ways to exit a running VM once certain CPU registers are accessed that can be incorporated to differ between native and virtual systems. In Section 3.6, we have discussed countermeasures that we have implemented in our current prototype.

Finally, a malicious piece of software could rearrange the code of the system libraries (or even the kernel) itself in a way that the region transitions do no longer correlate with intermodular calls. Obviously, our approach is incapable of dealing with such modifications since it is based on such region transitions. Nevertheless, besides full software emulators, all other existing malware analysis frameworks would also not be able to deal with code that modifies the system itself.

7 Conclusion

In this paper, we have proposed a new approach for hypervisor-based system monitoring using new hardware features. We have presented CXPInspector, a first prototype implementation of the proposed methodology that detects control flow transitions between user or kernel modules. This enables us to intercept intermodular function calls and their corresponding returns. Furthermore, it uses VMI to extract valuable data such as the used function parameters. We showed in several case studies different application scenarios for CXPInspector (e.g., malware analysis and spotting hot memory regions). The ability to trace current 64-bit kernel malware for Windows 7 (such as TDL4) is an outstanding effort of our system. To our knowledge, no other analysis framework exists that is capable to analyze such kinds of malware.

A Two-Dimensional Paging

Current x86/x64 architectures offer hardware support for virtualized memory in form of *Two-Dimensional Paging* (TDP). Though using different names for their implementation, the *Extended Page Tables* (EPT) used by Intel is conceptually similar to the *Nested Page Tables* (NPT) introduced by AMD. In the following, we focus on Intel's EPT mechanism but the basic principle also applies to NPTs. We explain the internal mechanisms behind EPTs based on Figure 10.

The main concept behind TDP is the establishment of one additional address translation layer between the physical memory seen by the guest and the real host physical memory. Before its

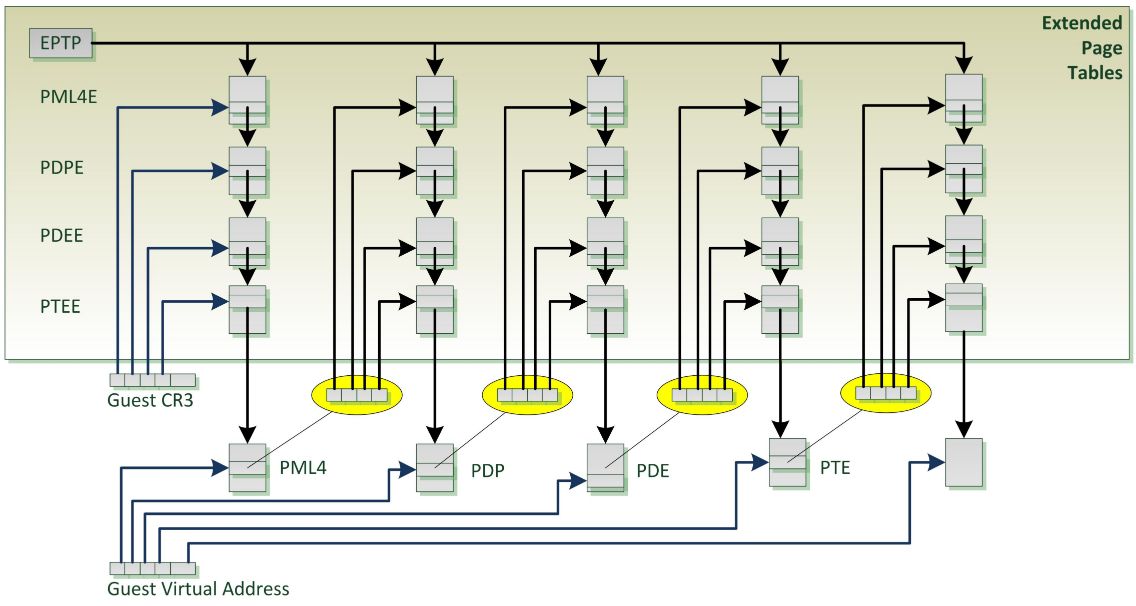

Figure 10: Two-Dimensional Paging

invention, hypervisors had to maintain *Shadow Page Tables* (SPT) for each hosted virtual machine to protect the host physical memory and offer isolated virtualized physical memory to each VM. To that end, the hypervisor enforced the virtual machines to use the SPTs instead of their regular page tables when virtual addresses had to be translated to physical ones. This happened completely transparent to the guest to generate the illusion that the regular page tables were indeed used. To realize this illusion, the maintenance of the SPTs had to be performed by the hypervisor and on each memory or page table access interception was required. For that purpose, many transitions between the VM and the hypervisor were necessary, which imposed a huge performance drawback.

To get rid of this performance penalty, TDP was introduced. Like shadow paging, TDP is also used for transparent address translation of the virtual machine. However, with TDP the guest uses its original page tables to translate virtual to physical addresses without any intervention of the hypervisor. At the time a specific physical guest memory address is actually accessed, one additional page walk takes place that translates a physical guest address to a physical host address. This second page walk is performed by the MMU without the assistance of the hypervisor. To find the corresponding host addresses, the MMU needs specific data structures for each virtual machine: the EPTs or NPTs. These structures resemble the regular page tables such that they are also organized in four levels. While each upper level contains the *Physical Frame Number* (PFN) of the next lower level, the last level contains the PFN of the resulting memory frame that corresponds with the originating guest frame. Note that all EPTs only contain host frame numbers, while all page table entries within the guest only contain guest frame numbers. Like regular page tables, also the EPT entries contain certain control bits beside the PFN. Most of them are similar to the flags of regular page table entries, i.e., dirty, accessed, and protection flags.

To further illustrate this mechanism, we describe an address translation from a guest virtual address (GVA) to a host physical address (HPA). For sake of simplicity we neglect the effects of *Translation Lookaside Buffers* (TLB), that cache address data to speed up the translation process. Note that even when considering the effects of TLBs, the first address lookup that fills the cache entries take place as described below.

Address Translation Example When the guest accesses some memory by a certain guest virtual address (GVA), the MMU first tries to resolve it to the corresponding guest physical address (GPA). To that end, the MMU determines the GPA of the highest guest paging structure entry (PML4) from the guest's CR3 register. In a non-virtualized system, the MMU would then read the PML4 contents from memory and proceed with the next level of the page walk. However, on a virtual system the GPA of the PML4 entry first has to be translated to a host physical address (HPA) to actually access its contents. This is done in the second dimension of the page walk, that — in contrast to the first dimension — only involves data structures maintained by the hypervisor. The first level of those host paging structures is called PML4E and is referenced by the *EPT pointer* (EPTP). Like in a regular page walk, the certain bit groups of the address to be resolved operate as index pointers into several extended page tables. After all of the multiple EPT structures are walked, the HPA of the guest CR3 is obtained and the PML4 of the guest can be obtained from memory. In the following, the MMU obtains the HPAs of all the other guest paging structures, each time with the help of a second dimension page walk like described above. Finally, the corresponding HPA to the initial GVA is obtained and the guest can finally access the memory.

Page Walk Faults Similar to a traditional page walk on a non-virtualized machine, also in the scenario described above page faults can occur during the page walk. The normal cause for these faults are non-initialized paging structures, since contemporary operating systems set them up in a lazy manner for performance reasons. When such page faults occur during the guest page walk, the regular page fault handler of the guest operating system is invoked and everything works as in a non-virtualized system. If the fault occurs while accessing the EPT entries, a dedicated VMExit is triggered (the exit reason is *EPT violation/misconfiguration*). The hypervisor then has to allocate physical host memory, set up a corresponding EPT entry, and, finally resume execution of the guest by performing a VMEnter. All of this happens transparently to the guest.

Acknowledgments

We would like to thank Jonas Pfoh, Christian Schneider, and Sebastian Vogel from the Technische Universität München, Germany, for their generous help while introducing us to KVM development and for the valuable discussions we had regarding our ideas. Furthermore, Johannes Hoffmann and Felix Schuster provided valuable feedback for an earlier draft of this paper.

This work was supported by the Ministry of Economic Affairs and Energy of the State of North Rhine-Westphalia (grant 315-43-02/2-005-WFBO-009) and the German Federal Ministry of Education and Research (BMBF grant 16BY1207B – iTES).

References

[1] T. Garfinkel and M. Rosenblum, "A Virtual Machine Introspection Based Architecture for Intrusion Detection," in *Symposium on Network and Distributed System Security (NDSS)*, 2003.

[2] P. M. Chen and B. D. Noble, "When virtual is better than real," in *USENIX Workshop on Hot Topics in Operating Systems*, 2001.

[3] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. T. Giffin, and W. Lee, "Virtuoso: Narrowing the Semantic Gap in Virtual Machine Introspection," in *IEEE Symposium on Security and Privacy*, 2011.

[4] Y. Fu and Z. Lin, "Space Traveling across VM: Automatically Bridging the Semantic Gap in Virtual Machine Introspection via Online Kernel Data Redirection," in *IEEE Symposium on Security and Privacy*, 2012.

[5] B. D. Payne, M. Carbone, M. I. Sharif, and W. Lee, "Lares: An architecture for secure active monitoring using virtualization," in *IEEE Symposium on Security and Privacy*, 2008.

[6] M. Carbone, M. Conover, B. Montague, and W. Lee, “Secure and robust monitoring of virtual machines through guest-assisted introspection,” in *Symposium on Recent Advances in Intrusion Detection (RAID)*, 2012.

[7] X. Jiang, X. Wang, and D. Xu, “Stealthy malware detection and monitoring through VMM-based ”out-of-the-box” semantic view reconstruction,” *ACM Transactions on Information and System Security (TISSEC)*, vol. 13, no. 2, 2010.

[8] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, “Ether: malware analysis via hardware virtualization extensions,” in *ACM Conference on Computer and Communications Security (CCS)*, 2008.

[9] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, “BitBlaze: A New Approach to Computer Security via Binary Analysis,” in *International Conference on Information Systems Security*, 2008.

[10] U. Bayer, C. Kruegel, and E. Kirda, “TTAnalyze: A Tool for Analyzing Malware,” in *Annual Conference of the European Institute for Computer Antivirus Research (EICAR)*, 2006.

[11] B. D. Payne and W. Lee, “Secure and flexible monitoring of virtual machines,” in *Annual Computer Security Applications Conference (ACSAC)*, 2007.

[12] D. Srinivasan, Z. Wang, X. Jiang, and D. Xu, “Process out-grafting: an efficient ”out-of-vm” approach for fine-grained process execution monitoring,” in *ACM Conference on Computer and Communications Security (CCS)*, 2011.

[13] D. Srinivasan and X. Jiang, “Time-traveling forensic analysis of vm-based high-interaction honeypots,” in *Security and Privacy in Communication Networks (SecureComm)*, 2011.

[14] J. Pfoh, C. Schneider, and C. Eckert, “Nitro: Hardware-based System Call Tracing for Virtual Machines,” in *International Conference on Advances in Information and Computer Security*, 2011.

[15] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, “Panorama: capturing system-wide information flow for malware detection and analysis,” in *ACM Conference on Computer and Communications Security (CCS)*, 2007.

[16] A. Vasudevan and R. Yerraballi, “Cobra: Fine-grained malware analysis using stealth localized-executions,” in *IEEE Symposium on Security and Privacy*, 2006.

[17] M. Egele, T. Scholte, E. Kirda, and C. Kruegel, “A survey on automated dynamic malware-analysis techniques and tools,” *ACM Computing Surveys (CSUR)*, vol. 44, no. 2, 2008.

[18] J. Wilhelm and T.-c. Chiueh, “A forced sampled execution approach to kernel rootkit identification,” in *Symposium on Recent Advances in Intrusion Detection (RAID)*, 2007.

[19] A. Moser, C. Kruegel, and E. Kirda, “Exploring multiple execution paths for malware analysis,” in *IEEE Symposium on Security and Privacy*, 2007.

[20] J. Rhee, R. Riley, D. Xu, and X. Jiang, “Kernel malware analysis with un-tampered and temporal views of dynamic kernel memory,” in *Symposium on Recent Advances in Intrusion Detection (RAID)*, 2010.

[21] J. S. Robin and C. E. Irvine, “Analysis of the intel pentium’s ability to support a secure virtual machine monitor,” in *USENIX Security Symposium*, 2000.

[22] D. Balzarotti, M. Cova, C. Karlberger, E. Kirda, C. Kruegel, and G. Vigna, “Efficient Detection of Split Personalities in Malware,” in *Symposium on Network and Distributed System Security (NDSS)*, 2010.

[23] M. Lindorfer, C. Kolbitsch, and P. Milani Comparetti, "Detecting environment-sensitive malware," in *Symposium on Recent Advances in Intrusion Detection (RAID)*, 2011.

[24] C. Willems, T. Holz, and F. Freiling, "Toward Automated Dynamic Malware Analysis Using CWSandbox," *IEEE Security and Privacy*, vol. 5, no. 2, 2007.

[25] C. Willems, R. Hund, A. Vasudevan, D. Felsch, A. Fobian, and T. Holz, "Down to the Bare Metal: Using Processor Features for Binary Analysis," in *Annual Computer Security Applications Conference (ACSAC)*, 2012.

[26] Intel Corporation, "Intel: 64 and IA-32 Architectures Software Developer's Manual," 2012.

[27] AMD, "AMD64 Architecture Programmer Manual, Volume 2: System Programming," 2012.

[28] KVM, "Kernel-based Virtual Machine (KVM)," http://www.linux-kvm.org/page/Main_Page, 2012.

[29] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen, "Detecting past and present intrusions through vulnerability-specific predicates," in *ACM Symposium on Operating Systems Principles (SOSP)*, 2005.

[30] C. Schneider, J. Pfoh, and C. Eckert, "Bridging the Semantic Gap Through Static Code Analysis," in *European Workshop on System Security (EuroSec)*, 2012.

[31] Frank Boldewin, "CSI Internet, Episode 4: Open heart surgery," http://www.h-online.com/security/features/CSI-Internet-Open-heart-surgery-1350313.html, 2011.

[32] M. Antonakakis, J. Demar, K. Stevens, and D. Dagon, "Unveiling the Network Criminal Infrastructure of TDSS/TDL4," Damballa Research Report, 2012.

[33] Qumranet, "Increasing Virtual Machine Density With KSM," KVM Forum, 2008.

[34] G. Miłós, D. G. Murray, S. Hand, and M. A. Fetterman, "Satori: enlightened page sharing," in *USENIX Annual Technical Conference*, 2009.

[35] C. A. Waldspurger, "Memory resource management in vmware esx server," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, Dec. 2002.

[36] Norman, "Norman Sandbox Whitepaper," http://download01.norman.no/whitepapers/whitepaper_Norman_SandBox.pdf, 2003.

[37] Michael Becher and Ralf Hund, "Kernel-Level Interception and Applications on Mobile Devices," University of Mannheim, Technical Report TR-2008-003, 2008.

[38] Claudio Guarnieri, "The Cuckoo Sandbox," http://http://www.cuckoosandbox.org, 2012.

[39] F. Bellard, "QEMU, a fast and portable dynamic translator," in *USENIX Annual Technical Conference*, 2005.

[40] L. Martignoni, R. Paleari, G. F. Roglia, and D. Bruschi, "Testing CPU emulators," in *International Symposium on Software Testing and Analysis (ISSTA)*, 2009.

[41] ——, "Testing System Virtual Machines," in *International Symposium on Software Testing and Analysis (ISSTA)*, 2010.

[42] T. Raffetseder, C. Krügel, and E. Kirda, "Detecting System Emulators," in *Information Security Conference (ISC)*, 2007.

[43] M. I. Sharif, W. Lee, W. Cui, and A. Lanzi, "Secure in-VM monitoring using hardware virtualization," in *ACM Conference on Computer and Communications Security (CCS)*, 2009.

[44] A. Lanzi, M. I. Sharif, and W. Lee, “K-tracer: A system for extracting kernel malware behavior,” in *Symposium on Network and Distributed System Security (NDSS)*, 2009.

[45] A. M. Azab, P. Ning, Z. Wang, X. Jiang, X. Zhang, and N. C. Skalsky, “Hypersentry: enabling stealthy in-context measurement of hypervisor integrity,” in *ACM Conference on Computer and Communications Security (CCS)*, 2010.

[46] C. Xuan, J. Copeland, and R. Beyah, “Toward revealing kernel malware behavior in virtual execution environments,” in *Symposium on Recent Advances in Intrusion Detection (RAID)*, 2009.

[47] Z. Deng, D. Xu, X. Zhang, and X. Jiang, “Introlib: Efficient and transparent library call introspection for malware forensics,” *Digital Investigation*, vol. 9, Supplement, no. 0, 2012.

[48] L.-K. Yan, M. Jayachandra, M. Zhang, and H. Yin, “V2E: Combining Hardware Virtualization and Software Emulation for Transparent and Extensible Malware Analysis,” in *ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, 2012.

[49] J. Nakajima, “Intel Updates,” Xen Summit November, 2007.

[50] C. Song, P. Royal, and W. Lee, “Impeding automated malware analysis with environment-sensitive malware,” in *USENIX Workshop on Hot Topics in Security*, 2012.

[51] 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.

[52] N. M. Johnson, J. Caballero, K. Z. Chen, S. McCamant, P. Poosankam, D. Reynaud, and D. Song, “Differential slicing: Identifying causal execution differences for security applications,” in *IEEE Symposium on Security and Privacy*, 2011.

[53] M. I. Sharif, A. Lanzi, J. T. Giffin, and W. Lee, “Impeding malware analysis using conditional code obfuscation,” in *Symposium on Network and Distributed System Security (NDSS)*, 2008.

[54] A. M. Dunn, O. S. Hofmann, B. Waters, and E. Witchel, “Cloaking malware with the trusted platform module,” in *USENIX Security Symposium*, 2011.

[55] J. Bethencourt, D. Song, and B. Waters, “Analysis-resistant malware,” in *Symposium on Network and Distributed System Security (NDSS)*, 2008.

[56] P. M. Comparetti, G. Salvaneschi, E. Kirda, C. Kolbitsch, C. Kruegel, and S. Zanero, “Identifying dormant functionality in malware programs,” in *IEEE Symposium on Security and Privacy*, 2010.

[57] C. Kolbitsch, E. Kirda, and C. Kruegel, “The power of procrastination: detection and mitigation of execution-stalling malicious code,” in *ACM Conference on Computer and Communications Security (CCS)*, 2011.

[58] J. R. Crandall, G. Wassermann, D. A. S. de Oliveira, Z. Su, S. F. Wu, and F. T. Chong, “Temporal search: detecting hidden malware timebombs with virtual machines,” in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.