A virtualization based system infrastructure for dynamic program analysis

Jiaqi HONG
Singapore Management University, jqhong.2015@phdcs.smu.edu.sg

A VIRTUALIZATION BASED SYSTEM INFRASTRUCTURE FOR DYNAMIC PROGRAM ANALYSIS

JIAQI HONG

SINGAPORE MANAGEMENT UNIVERSITY
2020

A Virtualization based System Infrastructure for Dynamic Program Analysis

by

Jiaqi HONG

Submitted to School of Information Systems in partial fulfillment of the requirements for the Degree of Doctor of Philosophy in Computer Science

Dissertation Committee:

Xuhua DING (Supervisor / Chair)
Associate Professor of Information Systems
Singapore Management University

Robert DENG Huijie (Co-Supervisor)
Professor of Information Systems
Singapore Management University

Debin GAO
Associate Professor of Information Systems
Singapore Management University

Fengwei ZHANG
Associate Professor of Computer Science and Engineering
Southern University of Science and Technology, China

Singapore Management University
2020

I hereby declare that this PhD dissertation is my original work and it has been written by me in its entirety.
I have duly acknowledged all the sources of information which have been used in this dissertation.

This PhD dissertation has also not been submitted for any degree in any university previously.

[illegible]

Jiaqi HONG

6 May 2020

A Virtualization based System Infrastructure for Dynamic Program Analysis

Jiaqi HONG

Abstract

Dynamic malware analysis schemes either run the target program as is in an isolated environment assisted by additional hardware facilities, or modify it with instrumentation code statically or dynamically. The hardware-assisted schemes usually trap the target during its execution to a more privileged environment based on the available hardware events. The more privileged environment is not accessible by the untrusted kernel, thus this approach is often applied for transparent and secure kernel analysis. Nevertheless, the isolated environment induces a *virtual address gap* between the analyzer and the target, which hinders effective and efficient memory introspection and undermines the correctness of semantics extraction. Code instrumentation mixes the analyzer code together with the target, thus they share the same execution flow as well as the virtual address space at runtime. The instrumentation code has native access capabilities to the target's virtual memory, which seamlessly introspects and controls the target. However, code instrumentation based schemes are inadequate to tackle malicious execution since the analysis can be detected, evaded, or even tampered with as noted in many recent works.

We securely bridge the virtual address gap by designing a system called the *Onsite Analysis Infrastructure*(OASIS) based on hardware virtualization technology. OASIS features a one-way address space sharing: on the one hand, the analyzer, as an independent full-fledged application, runs in a fused virtual address space comprising both its own space and the target's; on the other hand, the analyzer's space is separated and isolated from the target which still runs within its unmodified address space. We also extend OASIS with a significant instrumentation technique which allows secure transitions between the analyzer and the target without precip-

itating any CPU mode/privilege switch. In total, OASIS offers three capabilities to the analyzer: to reference the target virtual memory in the native way with mapping consistency; to dynamically control and instrument the target execution; and to transparently receive unmodified host OS services. With these capabilities, the analyzer performs *onsite analysis* on a malicious user/kernel thread running in the guest VM.

We propose two new dynamic analysis models based on OASIS: *Onsite Memory Analysis (OMA)* and *Execution Flow Instrumentation (EFI)*. In OMA, the analyzer examines the user/kernel thread's live virtual memory without interposing on its execution. We developed four tools to demonstrate its capability. The first one is a virtual machine introspection tool which is up to 87 times faster than the state of the art. The second one delineates the target's virtual memory layout without trusting any kernel objects. The third one captures the target's system call events along with their parameters without intercepting its execution. The last one generates the control flow graph for Just-In-Time emitted code. In EFI, the analyzer is provisioned with two options to directly intercept the user/kernel thread execution and dynamically instrument it. Despite being securely and transparently isolated from the target, the analyzer introspects and controls it in the same native way as the instrumentation code. We have also conducted three case studies. The first one is a cross-space control flow tracer which shows OASIS based EFI has better performance than existing hardware trapping based schemes. The second one works in tandem with Google Syzkaller which demonstrates EFI's agility in controlling and introspecting the target thread. The last one examines how a user-space program exploits the vulnerability in dynamically loaded kernel modules. EFI tools are well-suited for targeted and fine-grained analysis.

We have implemented a prototype of OASIS on an x86-64 platform and have rigorously evaluated it with various experiments including performance and security tests. OASIS and its tools remain transparent and effective against targets armed with anti-analysis techniques including packing.

Table of Contents

List of Figures

List of Tables

Acknowledgments

First and foremost, I would like to thank my supervisor and committee chair, Prof. Xuhua Ding, for his guidance all through my Ph.D. study. His rigorous logical thinking and research attitude makes me a better researcher. The discussion sessions with him are the greatest lessons I have ever had. Besides my supervisor, I would like to thank the rest of my dissertation committee members Prof. Robert Deng, Prof. Debin Gao, and Prof. Fengwei Zhang, for their effort and time in evaluating the thesis, and their comments in improving the thesis. I also thank my senior Siqi Zhao and junior Nguyen Hoang Minh for their collaborations, encouragement and friendship.

Chapter 1

Introduction

1.1 Problem Overview

Dynamic malware analysis systems collect and examine runtime intelligence from an untrusted target program, either by running it as is in a confined environment assisted by special hardware facilities such as Intel's Performance Unit(PMU) and ARM debug facility [1, 2], or modifying it with instrumentation code statically [3] or dynamically [4, 5]. Both approaches have their pros and cons.

The hardware-based schemes usually induce hardware events during the target execution. The event is trapped to a more privileged environment, e.g., the x86 VMX root mode [6, 7], the System Management Mode (SMM) [1] and the ARM Secure World [2]. These environments are not accessible to the kernel running in the Normal World or in a guest VM, thus this approach is often applied for secure and transparent kernel analysis. However, the security is attained at the price of convenience. Besides the non-flexible trapping mechanism which is restricted to available facilities provided by the hardware, the isolated environment also results in the so-called *virtual address gap* between the analyzer and the target as noted in [8]. The analyzer cannot directly read, write or execute in the target virtual memory.

The virtual address gap has several undesirable effects. One obvious drawback is the overheads of data collection and movement across different spaces, which is ill-suited for analysis demanding frequent and massive accesses to the target memory. A more subtle issue is that the analyzer has to explicitly handle the virtual address mappings used by the target, for instance, to dereference a pointer. This hassle not only complicates the analyzer's logic, but also affects the performance. More importantly, as in live forensics [9] and virtual machine introspection [10, 11], exported kernel objects (e.g., VMA structures and page tables) are heavily used to interpret and process virtual address related information. Hence, it cannot analyze malware with kernel privilege which fakes those dependent kernel objects or even stages an innocent-looking memory view. To bridge the virtual address gap is not as simple as it appears. Besides undesired side effects on code size and performance, it needs to deal with various kernel attacks such as translation redirection attacks [12], transient attacks [13, 14] and race condition attacks [15, 8].

Code instrumentation [16, 4, 17, 18] has been used in a myriad of software analysis applications and does not suffer from the aforementioned problem. Its hallmark is to integrate the analysis code and the target code into one binary either before execution or at runtime. Static code instrumentation inserts call back functions in the source code level or rewrites the binary. For example, Linux kernel provides KCOV function to collect code coverage information to support coverage guided kernel fuzzing tools like Syzkaller [19]. Dynamic binary instrumentation(DBI) [4, 20, 21, 22] re-compiles the target program's binary at runtime. The analysis code executes in the same virtual memory and CPU context as the target, a feature we denote as *native-access* in this dissertation. Hence, it seamlessly introspects and controls the target by directly referencing the latter's virtual addresses and accessing the registers.

However, recent research [23, 24, 25, 26, 27, 28] has shown that code instrumentation is inadequate to tackle malicious executions, since the analysis can be detected, evaded, and even tampered with. The culprit is exactly code-level integration. Sharing the entire address space and the execution flow becomes an attack surface. Although several mitigation techniques [26, 28] are proposed to improve instrumentation transparency and security, they do not tackle the problem at its root and cannot cope with kernel-level analysis at all. The approach of using the instrumentation code to prevent itself from being tampered faces the cyclic-dependence challenge and results in a complexly-engineered bulky program with a heavy performance toll. Besides the security and transparency issues, code instrumentation may acquire distorted intelligence due to target address space alternation and binary rewriting. Hence, the strengths and weaknesses of the code instrumentation are the opposite of those hardware-based isolation oriented approaches.

1.2 Research Objectives

In this dissertation, our research objectives are threefold.

Firstly, we aim to design a system infrastructure to securely bridge the aforementioned virtual address gap. The system provides the analyzer with the native access capability to a target program's virtual memory, as well as isolation and transparency protection. We aim to support a full-fledged user space analyzer, while the analysis target is an untrusted full-space thread.

Secondly, after securely bridging the virtual address gap, we aim to explore the intelligence recovered from the target program's live virtual memory which is underutilization in existing dynamic analysis schemes. The patterns and evolution of the virtual memory should be able to reveal the program's semantics and behaviors, such as system call events.

Finally and more importantly, we aim to design a system infrastructure that allows an analyzer to instrument a running user/kernel thread. The infrastructure combines the advantages of hardware-based approaches (strong security and non-intrusiveness) and code instrumentation (native-access) without their disadvantages.

1.3 Threat Model

We consider a commodity x86-64 multicore platform with CPU and MMU virtualization extensions. The platform is managed by a host OS (e.g., Linux KVM) occupying CPUs in the VMX root mode. In the rest of the dissertation, we also refer to the host OS as "the hypervisor" when the functionality is related to virtualization. The *target* is a running thread in the hardware-assisted virtual machine denoted as the *guest* in the dissertation. The guest software runs in the VMX non-root mode. We suppose that the analyzer is compiled into a position independent executable, i.e., no absolute addresses in its code or static data.

We trust the hardware, the firmware and the host OS. The adversary resides in the guest, possibly in the kernel. We consider attacks attempting to compromise the analyzer and/or detect its presence. Side-channel and denial-of-service attacks are out of scope. The analyzer may have memory vulnerabilities. It is orthogonal to our study to cope with memory corruption attacks [29] which exploit vulnerabilities in the analyzer's implementation by feeding poisonous data to the analyzer.

1.4 Background

The following two pieces of background knowledge are necessary to understand this dissertation. The first is the address translation with and without MMU virtualization. The second is the VMFUNC facility.

1.4.1 Address Translation and Virtualization

Address Translation without MMU Virtualization

Software on an x64 platform typically uses 48-bit virtual addresses (VAs) which cover 256 Terabytes in total [30]. The kernel occupies the higher half of the address space and leaves the other half for user space. The paging hierarchy on 64-bit platforms consists of four levels of paging structures. The top level is the Page Map

Level 4 (PML4) table whose base address is stored in the `CR3` register. The other three levels are the Page-Directory-Pointer-Tables (PDPTs), the Page-Directories (PDs), and the Page-Tables (PTs). Each page of these paging structures contains 512 entries indexed by 9 bits in a VA. An entry of a PML4, PDT or PD page stores the *physical address* (PA) of the next level paging structure page, while a PT entry points to the physical page mapped to the VA.

To translate a virtual address X, the MMU uses `CR3` to locate the physical location of the PML4 table in use. X's PML4 index is then used to locate PML4 entry that stores the physical address of the corresponding PDPT table. Similarly, X's indexes for PDPT, PD and PT are used one after another to finally locate the physical page mapped to X.

Address Translation with MMU Virtualization

When the platform enables MMU virtualization, the kernel in the virtual machine (a.k.a. the guest) still manages the four levels of paging structures as well as `CR3`. The MMU still looks up the entry at each level of the paging hierarchy using the four 9-bits indexes, respectively. The main difference is that *all* addresses in `CR3` and the four-level paging structures are *guest physical addresses* (GPAs), instead of host physical addresses (HPAs). To access a GPA, the MMU traverses the Extended Page Tables (EPT) to locate the corresponding physical address. Hence, to translate a virtual address, the MMU must consult the EPTs for four times in order to walk the PML4, PDPT, PD, and PT pages, and for the fifth time to get the physical page frame number. We illustrate the involvement of the EPT in address translation with Figure 1.1.

Hence, the EPTs play a pivotal role in choosing the next level paging structure page to use. Without modifying the PML4 entry, a change in the EPT mapping can redirect the MMU to a different PDPT page.

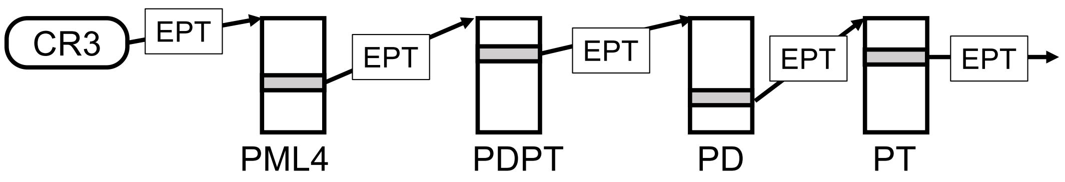


Figure 1.1: The EPTs are consulted for four times when the MMU visits four levels of paging structures and one more time for locating the final physical page.

1.4.2 VMFUNC

VMFUNC is an instruction provided under Intel's hardware Virtualization Technology(VT). It allows a non-root mode CPU to invoke multiple VM functions through specifying different indexes in the `RAX` register. The invocation can either come from the user or kernel space in a guest VM. Relevant controlling bits in the Virtual Machine Control Structure(VMCS) are set to enable the VMFUNC. By now, only one VM function with index 0x0 is implemented by Intel, which is to switch the Extended Page Table Pointer (EPTP) without causing a VM exit. The EPTP-switching VMFUNC is implemented as follows: the hypervisor prepares an EPTP list, each entry in the EPTP list points to a valid set of EPT paging structures. Based on the index specified in the `RCX` register, a `VMFUNC` instruction switches the underlying EPT paging hierarchy. The EPTP-list contains up to 512 8-Byte entries. Readers may refer to the Intel manual [30] for more details.

As reported in Table 6.1, a VMFUNC instruction costs as low as 147 CPU cycles. The overhead is even less a user-to-kernel switch, and much smaller than a VM transition. Thus, it has been adopted by many recent works [31, 32, 33, 34, 35, 36] to isolate different components within a guest VM with a low transition overhead. For example, SkyBridge [33] uses it to achieve a fast and secure inter-process communication, by using EPTs with different access permissions for those inter-process components.

TLB behaviors of EPTP Switching

We report three TLB behaviors of the EPTP-switching VMFUNC to help understand the performance and security of OASIS. First, the EPTP is associated with TLB entries. Programs running under different EPTPs cannot access each other, unless the underlying paging structures are the same. Second, TLB entries associated with different EPTPs can co-exist. This implies that an EPTP-switching VMFUNC does not flush TLB entries associated with other EPTPs, as long as there is enough space left in the TLB. Finally, an `INVLPG` instruction flushes the TLB entries associated with all EPTPs if they are related to the specified Virtual Address(VA) [37]. Our scheme uses this instruction to flush TLB entries of the target program when handling the self-modification code.

1.5 Dissertation Overview

In this dissertation, we first leverage the hardware Virtualization Technology(VT) to design and build a system infrastructure named **Onsite AnalySis InfraStructure(OASIS)** which securely bridges the virtual address gap. The system provides an *onsite environment* where a host analyzer application transparently executes within the target program's live virtual memory. The target's virtual address space is shared with the analyzer, but not vice versa. Thus the analyzer has native access capabilities to the target's memory as if accessing its own local buffer. The analyzer is a user space application which receives system services as usual when it introspects the malicious user/kernel thread.

Then we propose a new dynamic analysis model named **Onsite Memory Analysis(OMA)** based on OASIS. The analyzer examines the target's live virtual memory without interposing on its execution. As compared with existing out-of-VM introspection techniques, OMA has the following features. Firstly, it achieves *near-kernel introspection speed* with the differences only due to caching effects and code optimization. The high speed allows it to monitor system call issuances. Secondly, an OMA tool uses *zero-API* for introspection in the sense that the target virtual memory objects are referenced using their own virtual addresses as if they are in local buffers. Besides memory introspection, OMA also supports *virtual address testing* to reveal a page presence and potential pointers without software-traversing the paging structures or trusting any kernel objects. Lastly, existing static analysis tools can adapt to onsite mode to analyze Just-In-Time emitted code. We developed four tools to showcase the capabilities of OMA.

Finally, we extend the OASIS with a significant instrumentation technique to support a new dynamic analysis model named **Execution Flow Instrumentation(EFI)**. The extended system allows the analyzer running in the onsite environment to directly intercept the user/kernel target execution and dynamically instrument it in a native way. As compared with existing code instrumentation techniques [16, 4, 17, 18], the analyzer is guaranteed with isolation and transparency protection. As compared with existing hardware event-trapping based schemes [6, 7, 1, 2], the transition between the analyzer and the target does not entail any CPU mode/privilege switches such as VM exit. This not only saves the runtime CPU overhead, but also minimizes privileged software's involvement which leads to simpler analyzer development. With the one-way address space sharing, the analysis is conducted in a consistent execution environment with respect to the target's setting in the guest. Address related object dereferencing and instruction handling are more efficient. We conducted three case studies to demonstrate EFI's performance and agility in controlling and introspecting the target thread.

We implemented a prototype of OASIS (both for OMA and EFI) on an x86-64 platform and rigorously evaluated it with various experiments including performance and security benchmark tests. OASIS and its tools remain transparent and effective against targets armed with anti-analysis techniques including packing.

1.6 Dissertation Organisation

The remainder of this dissertation is organized as follows. Next, we review related works in Chapter 2. The OASIS infrastructure is described in Chapter 3. Chapter 4 presents the tools for OMA. The extensions on OASIS and the EFI techniques along with their case studies are discussed in Chapter 5. The implementation details and evaluations of OASIS are presented in Chapter 6. Finally, we discuss some of the issues in Chapter 7 and conclude the dissertation in Chapter 8.

Chapter 2

Related Work

We first discuss code instrumentation schemes and their security limitations in Section 2.1. Since our analysis target is a thread across user-to-kernel space, we briefly review kernel analysis frameworks in Section 2.2. Then we generally review the malware analysis and compare them with OMA in Section 2.3. The Virtual Machine Introspection(VMI) schemes are discussed in Section 2.4.

2.1 Code Instrumentation

Code instrumentation has been widely used in program analysis to collect their runtime information regarding system calls, API calls [38], control flows or data flows.

2.1.1 Static Code Instrumentation

Static code instrumentation operates either on source code or binary. The source code level instrumentation is normally used to enforce the program Control Flow Integrity(CFI) [39] or memory sanitizer [3, 40]. The instrumentation is done at compilation time through customizing open-source compilers such as GCC or LLVM. For example, the Linux kernel cooperates with GCC to insert Kernel Coverage(KCOV) and Kernel Address SANitizer(KASAN) code into the compiled kernel image.

Based on [41], binary-level code instrumentation is more challenging and generally uses two approaches: the direct modification and the full-translation. The direct modification approach [42] replaces the instrumentation point with the analysis code directly. This approach is restrictive since they should respect the original instructions' length as well as their semantics. The full-translation approaches [43, 44, 45] disassemble the binary first, transform it into an Intermediate Representation(IR), re-compile it with the instrumentation code, and finally store the newly generated binary on the disk. According to a study in 2016 [46], existing disassembler tools, such as IDA Pro [47], only detect less than 80% of function boundaries correctly due to multi-entry functions and tail calls [48, 49]. The study is conducted only on benign programs. The result should be much worse for malicious programs since they are intentionally equipped with anti-disassemble techniques [50].

2.1.2 Dynamic Binary Instrumentation(DBI)

Dynamic binary instrumentation (DBI) modifies the target along with its execution. Intel Pin [21], Valgrind [20], DynamoRIO [22] and Dyninst [51] are the best well-known versatile DBI frameworks. They normally recompile the target program's binary just in time with the instrumentation code and execute them in the tool's local buffer named code cache. Due to the sharing of the code as well as the address space, they are well-suited to collect runtime fine-grained memory information. For example, TaintCheck [52] uses Valgrind to make dynamic taint analysis. Instead of using just-in-time compilation and emulators, LiteInst [17] proposes to use instruction punning to insert probes into running code. More details can be found in DBI surveys [41, 28].

2.1.3 Security Limitations of Code Instrumentation

Many latest results [23, 24, 25, 26, 27, 28, 53, 54] have shown that sophisticated malware may detect the existence of DBI and therefore inhibits its malicious behaviors. Falc et. al. [55] release an open-source benchmark tool, named eXtensible Anti-Instrumentation Tester(eXait), to automatically test the existence of DBI frameworks on Windows platforms. PwIN [27] proposes a similar benchmark tool later named `jitmenot` [56] working on Linux platforms. Besides the detection, PwIN also demonstrates that it is easy to bypass the analysis of DBI by modifying the code in code cache directly since that memory region is RWX(readable, writable, and executable). SafeMachine [53] is the DBI tool targeted on malware analysis. But it only proposes patches for several well-known anti-instrumentation techniques, such as self-modifying code, memory relocation detection, and RIP context detection. It does not touch the essence of the problem which is the address space sharing. The security limitations are also acknowledged by the DBI tools. For example, DynamoRIO already makes some effort to preserve transparency by hiding its libraries from `EnumProcessModules()`.

As summarized in [27, 28], the security properties preserved by a DBI framework should include stealthiness and isolation. The stealthiness refers to that the instrumented application must not be able to infer the presence of the instrumentation platform; The isolation refers to that the instrumented application must not be able to tamper with the analyzing system including the analysis plugins. None of the existing DBI tools meets the two requirements. The security limitation is one of the motivations of our Execution Flow Instrumentation(EFI) analysis.

2.2 Kernel Analysis

Kernel analysis is more challenging than user space program analysis since the kernel is the system software managing the platform.

2.2.1 Code Instrumentation based Kernel Analysis

KernInst [57] allows the user to instrument any point of a running kernel by inserting an unconditional `jmp` instruction there. The `jmp` instruction jumps to a patch board where stores the analysis code along with the overwritten instructions. Our EFI analysis also uses the idea of the probe to transfer between the analyzer and the target, but our framework provides a strong isolation and transparency protection to the analyzer without suffering the convenience.

DBI has been applied to kernel analysis as well [18, 4, 58, 59, 5]. Cobra [16] uses DBI to analyze obfuscated code in both user- and kernel-mode. PinOS [4] extends Pin to instrument the Linux kernel on top of XEN [60]. PinOS steals memory inside the Guest VM and runs the instrumentation engine there, it is not security-oriented.

PEMU [5] is an out-of-VM DBI framework which aims to provide isolation between the instrumentation engine and the target VM. It prevents the direct attack from the guest. However, the instrumented code runs within the same address space of the analysis code, the attack surface remains there. Moreover, the out-of-VM solution introduces the so-called semantic gap [15] to the target VM. They follow HYPERSHELL [61] to redirect the inspection system calls back to the guest VM to bridge the semantic gap.

2.2.2 Hardware based Kernel Analysis

The other approach to kernel analysis is to leverage hardware features to trap the kernel execution into a more privileged system software. One of such hardware features is CPU and MMU virtualization. Gateway [62] uses a hypervisor to monitor how a driver invokes kernel API. Ether [6] is the first hypervisor-based framework for kernel analysis with the emphasis on transparency. SPIDER [7] achieves stealthy instruction-level tracing by using an instruction probe (i.e., INT3) to trap to the hypervisor. Intel SMM is used in MALT [1] to attain stronger transparency than relying on virtualization. Ninja [2] holistically explores a set of ARM features including TrustZone, PMU and Embedded Trace Macrocell (ETM), for cross-space debugging and analysis. kAFL [63] relies on Intel Processor Tracer to collect code coverage information for a coverage-guided fuzzer to find kernel vulnerabilities.

2.3 Malware Analysis

There are a vast literature on malware analysis including static analysis [64, 65], dynamic analysis [52, 66] and memory forensics [9, 67, 68].

Static analysis [64, 65] does not require execution of the malware. It conducts analysis on intelligences revealed by the static data or code (source code or binary) in order to understand the malware logic. The commonly used intelligence include control flow graph and call graphs. For instance, Eureka [69] is a tool that helps to construct annotated control flow graph of a packed malware. Major challenges of static analysis include malware's evasion capability by using obfuscation and self-modification, ambiguity of disassembly, unpredictability of data (e.g., destination of indirect transfer instructions). Moser et. al. summarized the limitation of static analysis in their work [70].

Dynamic analysis are based on intelligences extracted from a running software, instead of static text or data. According to a survey by Egele et. al. [71], dynamic malware analysis techniques can be broadly grouped into function call monitoring [38], function parameter analysis, information flow tracking (e.g., taint analysis [52]), instruction trace [72] and autostart extensibility points. OMA is one of the dynamic analysis techniques since it also inspects a running software. Different from existing schemes, OMA focuses on the target software's virtual memory, instead of directly upon runtime activities. Nonetheless, runtime activities leave traces in the memory. OMA provides a broader scope of intelligence collection and can be used to guide behavior analysis which is often implemented in an intrusive manner.

OMA is also relevant to memory forensics [9, 67, 68]. Existing memory analysis techniques require plenty physical memory dumps of the target. Additional computation has to be applied in order to re-construct the virtual-to-physical address mapping semantics. They target on known memory pattern signatures or rely on differential analysis to recover the malwares critical behaviors. There are two shortcomings. Firstly, the analyzer tool cannot use the real-time address mappings to translate virtual addresses to physical addresses. It therefore faces the thorny semantic-gap problem [15]. Secondly, it costs time to take memory snapshots. Once a copy is taken, it may become stale as it is no longer updated by the target. Therefore, it is more suitable for post-mortem analysis. OMA ensures mapping consistency and the data fetched on-demand is ensured to be fresh.

	Intelligence			**Acquisition**			**Best for extracting semantics of**
	Origin	**Runtime Events**	**CPU context**	**Intrusive**	**Operation**	**Timing**	
Static Analysis	binary image	✗	✗	N/A (the target does not run)	file read	N/A	logic of static code
Dynamic Analysis	virtual memory & CPU	✓	✓	YES	interception, hooks, instrumentation	event trigger	program/system runtime behaviors
Memory Forensics	physical memory	✗	✗	YES (in most cases)	memory dump	periodic or event triggered	data objects
OMA	virtual memory	system calls	✗	NO	native memory access	on demand	runtime virtual address space

Table 2.1: Comparisons between static analysis, dynamic analysis, memory analysis and OMA.

Comparisons. All four types of analysis tap into intelligence sources in order to gain understanding of various aspects of the target. Due to distinct intelligence types, they differ in terms of the intelligence acquisition method, the type of knowledge extracted. A detailed comparison is presented in Table 2.1.

2.4 Virtual Machine Introspection

Another related area is virtual machine introspection, especially out-of-VM introspection such as process out-grafting [73], Virtuso [10], VM space-traveling [74], Hybrid-bridge [11], and ImEE [8]. With out-of-VM introspection techniques, a monitor program inspects the target VM memory from the outside. As compared to in-VM introspection schemes (e.g., SIM [75]) relying on a trusted module in the target kernel, out-of-VM introspection is more secure and easier to deploy.

Among them, ImEE is the closest to OASIS. It tackles the semantic-gap problem [15] by using the target paging structures outside of the guest to attain mapping consistency. It allows a carefully crafted agent with dozens of instructions to making native accesses to the target virtual memory. Nonetheless, the agent does not have its own address space and cannot make system calls. Hence, ImEE can only be used as a memory access engine for introspection. It is infeasible to replace the agent with a full-fledge application.

Chapter 3

OASIS Infrastructure

In this chapter, we propose a new dynamic analysis paradigm called *onsite analysis* whereby the analyzer program securely runs within the malware's sample's live virtual address space. With the space co-residence, the analyzer has native access to the target's virtual memory. The address space sharing is one way in the sense that the target is prevented from detecting or accessing into the analyzer's space. To realize onsite analysis, we leverage the hardware virtualization technology to build a novel system named **Onsite Analysis Infrastructure(OASIS)**. The system allows a full-fledged analyzer program in the host to securely and transparently execute in the live virtual address space of the target program running in a hardware-assisted virtual machine. OASIS makes no modifications to the analyzer's binary, the target's binary, or the target environment including the libraries and the OS.

CAVEAT: The OASIS infrastructure presented in this chapter only allows the analyzer to conduct onsite memory analysis. The target program runs within the guest VM. The analyzer does not interpose the target's execution. Later in Chapter 5, we present the extensions on OASIS to allow flexible and secure instrumentation on the target execution.

Organization. In the next section, we propose the notion of onsite analysis and explain the high-level approach of OASIS. We then elaborate the design details of OASIS in Section 3.2.

3.1 Onsite Analysis and OASIS

This section elaborates the notion of onsite analysis and sketches out the design of OASIS that provides the system support for software performing onsite analysis.

3.1.1 Onsite Analysis

The key feature of onsite analysis is the one-way address space sharing, where the analyzer has the same view of the target's virtual memory as the target itself, but not vice versa. Mapping the target's physical memory into the analyzer's space does *not* realize onsite analysis, because only the physical memory is shared, instead of the virtual memory.

Implementing onsite analysis is non-trivial. Although Zhao et. al. [8] recently proposed ImEE which allows a special segment of instructions to run onsite, it is far from capable for onsite analysis. The reason is that ImEE imposes strict restrictions on the code running onsite: all instructions must reside in one page and can only access the target's address space; no data section is allowed and no OS service is provided. Below are two major challenges to enable practical onsite analysis.

Address Space Layout Matching. The foremost challenge is to fit the precompiled analyzer binary to the live address space of the target which is not known beforehand. Any solution must take the runtime dynamics of the target address space into consideration.

OS support. Like other programs, the analyzer execution demands OS services such as memory allocation and deallocation, file operations, and inter-process communications. Since the analyzer runs in the target's space, it is challenging to preserve a consistent memory view between the analyzer and the OS.

3.1.2 Overview of OASIS

We design OASIS as the infrastructure to support a pre-compiled executable to perform onsite analysis. Its high-level architecture is showed in Figure 3.1. The software components comprise a *trampoline* and the *OASIS manager*, both of which reside within the host OS as shown by the gray regions in Figure 3.1. The analyzer thread lives in two exclusive modes: *onsite mode* for the analyzer execution and *host mode* for the host OS execution in the same thread context. Hence, onsite/host mode is corresponding to user/kernel mode of a regular thread. In onsite mode, the analyzer performs onsite analysis with its CPU in the VMX non-root mode and a proper set of EPTs prepared by the OASIS manager. With the support of paging structures, it smoothly accesses both its own virtual memory (managed by the host OS) and the target's. Whenever the OS service is needed in onsite mode, e.g., due to system call issuance or exceptions, the mode is automatically switched to host mode via the trampoline. Onsite mode is transparent to the host OS so that the analyzer's events are handled as if they are from the analyzer thread in host mode.

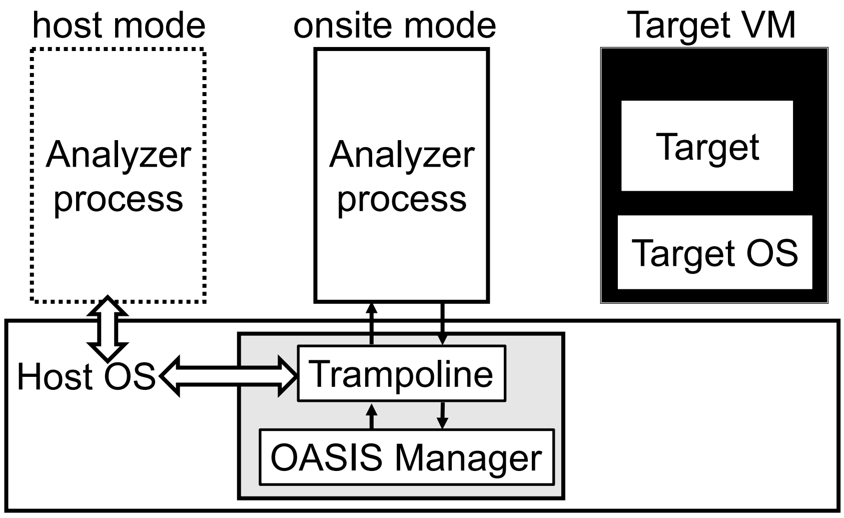


Figure 3.1: The architectural view of OASIS. The whole target VM is untrusted. The analyzer runs in onsite mode to perform onsite analysis upon the target while the host OS treats it as a normal user-space process.

The cornerstone of OASIS is the paging structures used in onsite mode. We sketch out the main design of the paging structure below and leave the full details in the next section.

Main Design. Let $\mathbf{A}$ denote the set of virtual addresses (VAs) used by the analyzer in onsite mode. Since the analyzer makes both remote accesses to the target and local accesses to its own code and data, $\mathbf{A}$ is split into $\mathbf{A}_l$ and $\mathbf{A}_r$ (namely $\mathbf{A} = \mathbf{A}_l \cup \mathbf{A}_r$) where $\mathbf{A}_r$ and $\mathbf{A}_l$ denote VA sets for remote and local accesses, respectively. Note that $\mathbf{A}_l \cap \mathbf{A}_r = \emptyset$ because no VA is used to reference a local object and a target object. The paging structures for onsite mode ensure mapping consistency and data consistency:

- **cross-domain consistency.** $\forall A \in \mathbf{A}_r$, the MMU servicing the analyzer in onsite mode translates A to the same physical address as the target VM's MMU does. Cross-domain consistency implies the correctness of the accessing the target virtual address space.

- **cross-mode consistency.** $\forall A \in \mathbf{A}_l$, the MMU servicing the analyzer in onsite mode translates A to the same physical address as it is mapped to in host mode. Cross-mode consistency is the prerequisite for the host OS to provide the correct service to the analyzer, since the host OS and the analyzer have the same view on the userland.

To attain cross-domain consistency for $\mathbf{A}_r$, we clone the target's PML4 page to onsite mode's PML4 page, denoted as O-PML4. The GPA-to-PA mappings defined on the target EPTs are copied over to onsite mode, with all execute permissions being removed. Due to the synchronization between O-PML4 and the target's PML4 page, the VA-to-PA mappings for $\mathbf{A}_r$ are exactly the same as the target's.

To attain cross-mode consistency for $\mathbf{A}_l$, our high-level approach is to graft the analyzer's host PDPT page into the paging hierarchy in onsite mode. The analyzer's Page Directory (PD) and Page Table (PT) pages in host mode are mapped to the onsite mode by using the identity GPA-to-PA map. Hence, all of the following paging structures in the host starting from PDPT are automatically grafted to the hierarchy in onsite mode, all entries thereof remain valid. By grafting one PDPT page only, the analyzer's local 512 GB virtual address space is fused into the target's address space. Our approach does not modify the analyzer's host paging structures. Cross-mode consistency is achieved because the host OS and the onsite analyzer use the same paging structures starting from the PDPT page.

CAVEAT. We acknowledge that detection of virtualization remains possible. Nevertheless, the presence of virtualization alone is not decisive enough for future malware to inhibit its malevolent behaviors. Recent years have seen a surge of virtualization support in computing platforms including mobile phones [76, 77] and a wider adoption by commodity operating systems including Microsoft Windows 10.

3.2 Main Design of OASIS

In this section, we begin with elaboration of the analyzer thread virtual address space layouts in both host mode and onsite mode. We then present in details how $\mathbf{A}_l$ and $\mathbf{A}_r$ are translated properly in onsite mode, followed by the description of mode switches.

3.2.1 Analyzer's Address Spaces

The 48-bit virtual addresses on an x64 platform cover 256 Terabytes in total [30]. As shown in Figure 3.2, the leading 9 bits of a VA serve as the index to the PML4 table during address translation. For convenience of expression, we call it the *PML4 prefix* of a virtual address.

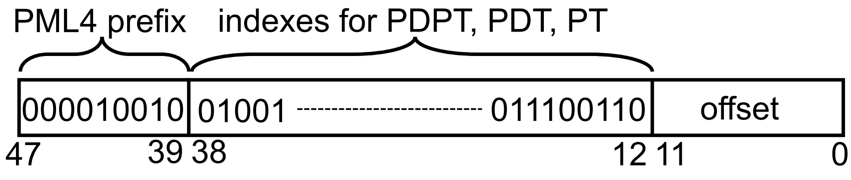


Figure 3.2: Illustration of a 48-bit virtual address on a 64-bit platform. Its PML4 prefix (namely bit 39 - 47) is "00001010".

Host Mode. The host OS manages the analyzer's address space. OASIS imposes a restriction that all virtual addresses of the analyzer have the same PML4 prefix (denoted as λ_h). The implication is that all VAs are in a 512GB region and are translated via one PDPT page pointed to by one PML4 entry. This facilitates grafting the PDPT page. Since the analyzer does not execute in host mode, only the host OS accesses its pages using VAs with the PML4 prefix λ_h.

Our scheme does not require the host OS to change the layout of the analyzer's address space, i.e, the distances among its stack, heap, and various data/code sections.

Onsite Mode. If the analyzer's VA regions allocated by the host OS are directly used as $\mathbf{A}_l$ in onsite mode, it may collide with $\mathbf{A}_r$. To cope with this issue, we relocate it to a 512 GB space which is unoccupied by the target program. Note there are 512 512 GB band, most of which are not empty. We denoted the PML4 prefix of the chosen VA range as λ_o. It defines $\mathbf{A}_l$. The physical pages allocated by the host OS to the analyzer are not moved. Hence, if the analyzer physical page is accessed by the host OS using a VA in the form of "$\lambda_h||$offset", it is visited by the analyzer in onsite mode using the VA in the form of "$\lambda_o||$offset".

Address relocation obviously breaks cross-mode consistency, if there is no other treatments. Section 3.2.3 explains how to cope with the discrepancy of the prefixes to withhold cross-mode consistency, so that the relocation is transparent to both the host OS and the analyzer.

3.2.2 Paging Structures in Onsite mode

The definition of onsite analysis mandates that only one set of paging structures are used in onsite mode to map all VAs in $\mathbf{A}_l \cup \mathbf{A}_r$. OASIS provides the needed paging structures without requiring the compiler to differentiate remote and local addresses or the program developer to explicitly declare them.

Guest Page Table. The guest page tables, which translate VAs to GPAs, are depicted in Figure 3.3 below. It is rooted at O-PML4, followed by the target's paging structures in the guest to access $\mathbf{A}_r$, and the analyzer's paging structures in the host to access $\mathbf{A}_l$. We use an unoccupied entry on O-PML4 and populate it with a randomly generated GPA β, it is redirected in the onsite mode EPT which guides the MMU to the analyzer's host PDPT page when it translates β. If we denote the guest's GPA domain as $[0, T]$, T is determined based on the guest VM's configuration. β and the onsite CR3 are randomly chosen from $[T, 2^{48}]$ to avoid conflict GPA usages with the target VM. With the assistance of the EPTs, the analyzer's host mode paging structures starting from the PDPT page are fused into onsite mode paging hierarchy.

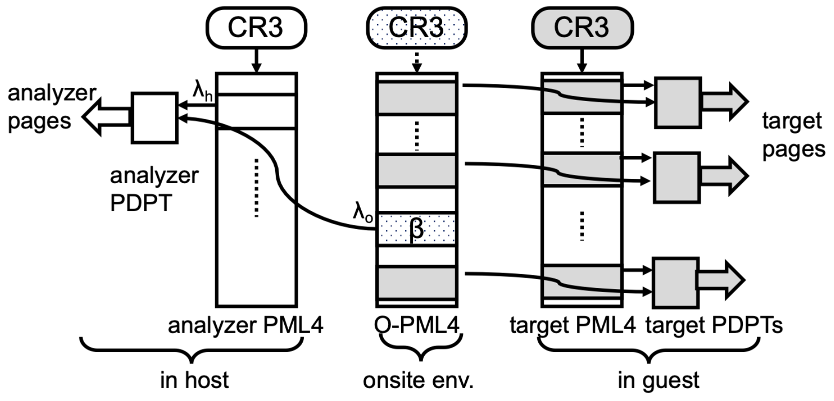


Figure 3.3: Overview of onsite guest paging structures. The analyzer's PDPT page in host mode is grafted into onsite mode through the O-PML4 entry β with PML4-prefix λ_o. All arrows except those in the host mode involve the underlying GPA-to-HPA mappings in onsite mode EPT.

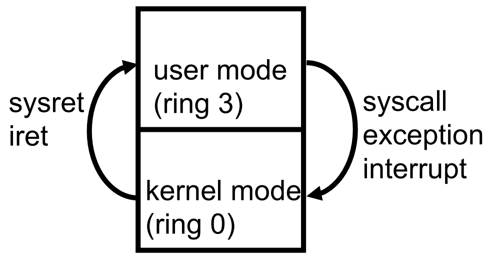
EPT. The analyzer's onsite execution requires physical pages storing its code and data sections, the stack, the heap, and the paging structures. All these pages are allocated by the host OS and their physical addresses are used directly as the corresponding GPAs (except the PDPT page). In other words, the EPTs for local GPAs define an identity map. The host OS allocates those physical pages from the page pool between T and the maximum physical memory on the platform. The page pool has no overlap with the target's GPA domain thus no conflict GPA usages with the target in onsite mode.

The EPTs also comprise mappings copied from the target VM to ensure cross-domain consistency. Namely, if a GPA is used by the target VM, the EPTs map it into the same physical page as in the target VM.

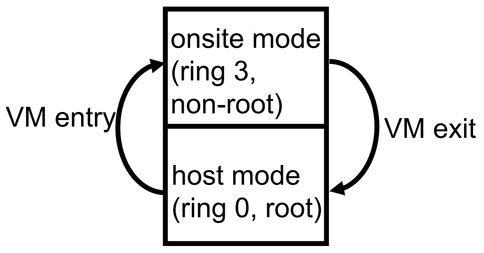
To summarize, the guest page tables and EPTs presented above, as well as the aforementioned address space allocation, ensure cross-domain and cross-mode consistency simultaneously.

3.2.3 A Thread of Two Modes

A normal thread runs in either user-mode or kernel-mode and the CPU privilege changes along with mode switches. Switches between host mode and onsite mode in OASIS involves both the CPU privilege change and operation mode change (root v.s. non-root), as shown in Figure 3.4.



(a) Normal thread privilege switch



(b) Analyzer thread mode switch on OASIS

Figure 3.4: Comparison between normal mode switch and OASIS's mode switch

The analyzer in onsite mode switches to host mode via VM exit which is triggered by system calls, excepts and interrupts, the same events that trigger privilege switches in a normal environment. In host mode, the host OS handles the event and resumes the analyzer thread execution in onsite mode via VM entry. Since neither the hardware nor the host OS directly supports such mode switches, we introduce the trampoline and the OASIS manager modules in the host OS to handle them. The workflow of mode switches is illustrated in Figure 3.5. Switches between host and onsite modes do *not* change the thread context (i.e. no task switch incurred), which, jointly with cross-mode consistency, ensures correct kernel services to the analyzer events.

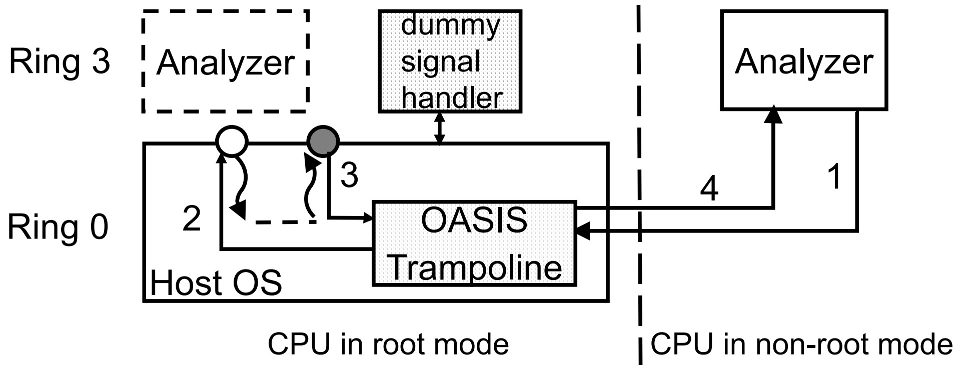


Figure 3.5: Mode switches for the host OS to handle events in onsite mode. The white circle denotes the entry of the kernel event handler while the black circle denotes the user space entry for the kernel to return.

Onsite-to-Host Switch

The main task for switching to host mode from onsite mode is to adapt the context produced by the hardware during VM exit into the context that would have been produced by the hardware during a privilege switch for the same event. When a VM Exit is triggered (Step 1 in Figure 3.5), the hardware saves the analyzer's vCPU registers into the VMCS structure, switches the CPU mode to VMX root mode, and passes the control to the trampoline which is priorly set as the default VM exit handler for the analyzer.

System calls and interrupts are simple to handle because only registers are involved. Their occurrences in onsite mode trigger a VM exit because the absence of kernel handlers leads to an EPT violation, which does not affect those registers. Hence, the trampoline makes no adaption except loading the physical CPU's registers according to the VMCS structure.

It is slightly more complex to adapt the context for exceptions. When handling a user-space exception in a normal setting, the hardware uses the kernel stack and some registers to provide the necessary data for the kernel to resolve the issue. A stub handler is installed in onsite mode to capture the exceptions and save contexts. It then issues a hypercall to notify the trampoline. The trampoline follows the hardware convention to fill in the kernel stack and load relevant registers based on the saved context.

Once the event context is ready, the trampoline invokes the corresponding OS handler in a returning-boomerang fashion. It first sets a hardware breakpoint to intercept the kernel's return to user-mode of the analyzer thread, and then *jumps* to the entry of the kernel handler (Step 2). With a properly prepared event context and cross-mode consistency, the host OS can smoothly attend to the event in question. Unless the analyzer thread is aborted, the kernel eventually attempts to return to the user mode of the thread, instead of onsite mode. Hence, the breakpoint hijacks the control flow right after privilege switch so that the trampoline regains the CPU and returns to onsite mode.

Signal handling requires a special treatment since the kernel needs to call the user-space signal handler if any. To maintain the semantics of signals, a dummy user-space signal handler is registered to the OS if the analyzer program has any non-default signal handler. When invoked by the OS, the dummy handler saves the parameters (if any) and sets up a flag to indicate its invocation. In this case, the actual non-default handler is invoked after the analyzer resumes in onsite mode.

Host-to-Onsite Switch

Entering to onsite mode is always triggered by the debug exception upon the previously set breakpoint (Step 3). The trampoline removes the breakpoint immediately, because its execution will be carried out under the analyzer thread context no matter if it is interrupted or not. The trampoline calls the OASIS manager to prepare the vCPU and EPTs for onsite mode.

The first time entrance requires the OASIS manager to set up the Virtual Machine Control Structure (VMCS), O-PML4 as well as the EPTs as described in Section 3.2.2. The `RIP` register is loaded with the address of the first user-space instruction of the analyzer process in onsite mode.

For returns following a VM exit, the OASIS manager needs to update the saved register context in VMCS if it returns from a system call handling, including `RIP`, RAX and RFLAGS. Then the OASIS manager uses the updated context to resume the analyzer thread. Besides that, it also checks whether the analyzer's paging structures in host mode are updated by the kernel. Exceptions such as page faults may entail address space changes. For new physical pages added to the paging hierarchy or the analyzer's space, the OASIS manager updates the EPTs with new identity mappings.

In the end, the trampoline uses VM-entry instructions to switch the CPU to VMX non-root mode and the hardware runs the instruction pointed to by the RIP in the VMCS (Step 4), which launches/resumes the analyzer in onsite mode.

Address Adjustment

The host OS's service to the analyzer's event may result in new data being written into user-space buffers, e.g., the I/O data generated by a peripheral device is copied to a user-space buffer chosen by the analyzer. It is therefore crucial to maintain cross-mode consistency so that returns from the OS is properly received by the analyzer.

Since the analyzer's VAs have different PML4 prefixes (i.e. λ_h and λ_o) in host and onsite modes, the trampoline handles this difference during mode switches. When entering host mode from onsite mode, it replaces λ_o with λ_h in all addresses passed to the host OS such as system call parameters in general registers and the address of the faulting instruction in CR2. When entering onsite mode, it replaces λ_h with λ_o in all addresses returned to the analyzer, such as the new program break from a brk system call. Note that the host kernel's update on the analyzer's paging structures does not require address adjustment since they are physical addresses.

3.2.4 Security

The adversary we consider resides in the target VM with kernel privileges. Since onsite mode uses its dedicated O-PML4 page, no artifacts are exposed to the guest VM. The guest VM cannot manipulate the pages used by the analyzer since those pages are not mapped in the EPTs used in the guest VM. Transparency and isolation protection is guaranteed. Note that we do not consider memory corruption attacks (e.g., buffer overflow) that exploit the analyzer's software vulnerability, as the countermeasure [29] is orthogonal to our study. Onsite analysis is not more susceptible to such attacks than other analysis models.

3.3 Summary

In this chapter, we proposed OASIS as the system infrastructure to provide the onsite environment which guarantees the mapping consistency as well as native access speed. Its main idea is to fuse the target program's virtual address space within the analyzer's local address space through page table grafting assisted by EPT. We also addressed the technique challenges induced by the address space relocation for the analyzer to transparently receive system services from the host OS. The following two chapters discuss the two new dynamic analysis models benefiting from OASIS's native access capability, transparent system services as well as the isolation and transparency protection. The implementation and system evaluation are left in Chapter 6.

Chapter 4

OASIS based Onsite Memory Analysis

In this chapter, we propose a new dynamic analysis model named **Onsite Memory Analysis (OMA)**. In OMA, the analyzer examines the target by conducting native accesses to its live virtual memory without intercepting its execution. Existing memory-based dynamic analysis such as [78] is essentially based on physical memory dumps. Additional computation has to be applied in order to reconstruct the semantics in the virtual memory, which incurs not only performance overhead but also security risks. We remark that OMA differs from traditional memory analysis in several ways. The most prominent difference is that the latter relies on intermittent page-level memory snapshots, while onsite memory analyzer makes continuous and word-granularity memory accesses. Due to the operational difference, the latter makes postmortem analysis on the target's momentary state while OMA is fresh and state. More comparisons are in Chapter 2.

We developed four exemplary tools to showcase the capabilities of OMA. The first one is a Virtual Machine Introspection(VMI) tool, which achieves near-kernel introspection speed and references the target memory directly without calling any VMI APIs. The second one reveals the target's virtual memory layout without relying on any kernel objects. Existing approaches typically follow the conventional address space layout or trust the VMA structures provided by the target kernel [15]. Both methods are exaggerated and fail to work when malware reshapes its address space or subverts the target kernel. It also discovers potential pointers and link lists on the memory. The third one monitors the evolution of a particular memory region to capture the target's system call events as well as their parameters. Lastly, we apply an existing static analysis tool to generate control flow graph for Just-In-Time code. It demonstrates that existing static analysis tools (e.g., CFG generator) can be adapted for OMA.

Organization. The four tools are present in Section 4.1, 4.2, 4.3, and 4.4 respectively

4.1 Virtual Machine Introspection

Since the analyzer's address space encloses the subject's memory introspection is as natural as referencing its own memory buffer. The analyzer does not need any API to request OASIS services at runtime. Hence, the analyzer development is much easier than on other popular VMI schemes such as LibVMI [79] and XenAccess [80]. We show a toy C program below to demonstrate that the introspection tool on top of OASIS is easy-to-develop. Then we develop several typical memory introspection tools that fetch kernel objects from the target guest VM and make the performance comparison with ImEE [8], which represents the state of the art out-of-VM introspection technique. According to our experiments reported below, onsite memory introspection is up to 87 times faster then ImEE.

Toy Introspection Program on OASIS

The program showed in Figure 4.1 uses a target's Virtual Address(VA) to read the target's live memory content directly, and saves the result into its local file. The application requests OS services as usual and does not need any VMI APIs.

```
void main(){
  // an address in guest
  void* target_addr;
  target_addr = 0xffffffff816f3090;

  //open a local file in host
  int fd = open("dump.bin", O_RDWR);
  //copy 8 bytes from the guest
  write(fd, target_addr, sizeof(long));
  close(fd);
  return;
}
```

Figure 4.1: A toy analyzer that copies 8 bytes from the target's memory to a local file.

Performance Comparison

The available ImEE implementation and OASIS use 32-bit and 64-bit guest respectively, so the introspection performance is measured as the ratio of the analysis tool's CPU time to the time spent by the respective kernel module performing the same task. Figure 4.2 reports the performances for reading a consecutive block from the guest, which reflect the speed in accessing the target's memory. Our tool is about 9 to 87 times faster than its ImEE counterpart.

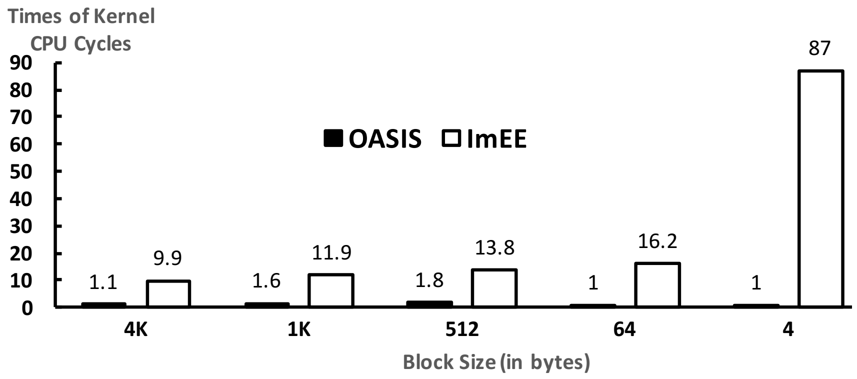


Figure 4.2: Tools using ImEE and OASIS have different performance in reading a block of memory. The smaller number represents higher performance.

Figure 4.3 reports the performances for reading through a list, which reflects the flexibility of the introspection tool's reading pattern. Among three analysis tools we test, the first two have fixed reading pattern, while the last one does not.

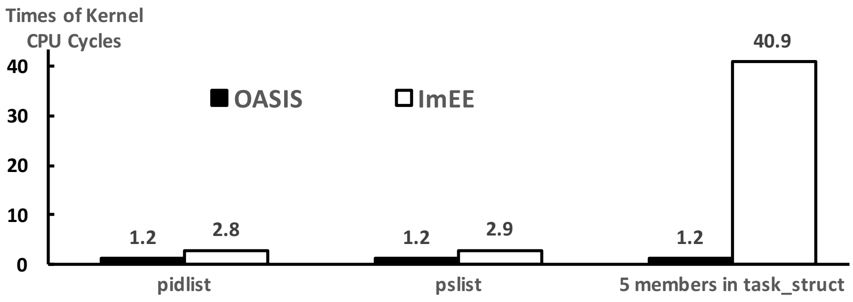


Figure 4.3: Traverse-read CPU time normalized by the kernel expenses for the same tasks. The smaller number represents higher performance.

Our tools also outperforms their ImEE counterparts with a 40 times gap for the last tool. The main reason is that, since the read pattern is not fixed, the ImEE tool cannot make a batch read as in the other two cases. It incurs significantly more synchronized reads and writes via the interface between the ImEE agent and the tool itself.

4.2 Virtual Address Space Reconnaissance

The tool discovers the target's virtual memory layout as well as potential pointers in the virtual memory and their values. An analysis of the pointers further unearths link-lists used by the target.

4.2.1 Design Sketch

At the core of the tool is an execution based *pointer-test*. The tool treats every eight bytes in the virtual address space as a virtual address P, and uses a `mov` instruction to read another eight bytes at P from the target memory. If no exception is thrown out, the bytes are a pointer which has the memory mapping. More precisely, the test shows that these bytes are *qualified* to be a pointer. Further analysis is needed to determine whether they are actually used as a pointer by the target program.

The reconnaissance tool makes a width-first like search starting with a given virtual page. It first *expands* the territory by reading lower and higher pages until encountering a page fault. It then *explores* these newly discovered virtual pages by running the pointer test to discover pointers at all possible virtual addresses. If those newly discovered pointers point to pages which have not been visited, the program appends these new pages to the queue for future expansion. The expand-then-explore cycle repeats until no new virtual pages are found. At the end, all memory pointers in all discovered pages are unearthed. By analyzing the relationship among pointers, the program can discover existence of link-lists and frequently used data objects.

There are several alternatives to obtain the starting page. A straightforward choice is to choose a random page or choose one from the default layout. Better options are to select the pages based on the current execution context. One is the target's stack page. When the target VM is trapped to the hypervisor, its `RSP` value can be read from its VMCS structure. Since stack frames contain return addresses, the program also finds out the code pages that are currently in execution. Another alternative is to find possible virtual addresses stored in general registers during VM exit or the VA of the intercepted instruction.

4.2.2 Implementation

Range checking is the popular method used to decide whether the given eight bytes are a valid VA or not. They are compared against the known VA ranges whose information is extracted from the kernel. As compared to the range-checking approach, the pointer-test approach is much faster for genuine pointers. However, it costs more CPU cycles for non-pointers due to the exceptions. Our implementation is optimized to minimize the cost of exceptions.

The first optimization is to filter out non-canonical addresses, namely between 0x7FFFFFFFFFFF and 0xFFFF800000000000, which are considered as illegal by the hardware implementing 48-bit addresses. The second optimization is a three-pronged approach to reducing the overhead of handling page faults triggered by accessing unmapped addresses. Firstly, we configure the onsite mode GDT so that the tool runs with Ring 0 privilege, instead of Ring 3. It avoids expensive privilege switches when page faults are thrown out. Secondly, a special page fault handler is installed for the tool. The handler's main task is to set a flag and advances the test according to the step length. The IDT is configured to disable stack switches for page faults. Thirdly, different from normal handlers that use `iret` to resume the faulting instruction, it uses `ret` instruction to make faster control transfer. The total implementation consists of 446 SLOC.

4.2.3 Experiment

We apply the address space reconnaissance tool upon Apache and Ustealer[1] which is a malware stealing Ubuntu information. Starting from the stack page, the tool gradually discovers the target's virtual pages and all potential pointers therein. We use the kernel's report on the address space as the baseline. Specifically, we count the number of mapped pages by checking the page table. We also use the VMA structures in the kernel to *calculate* the mapped pages. The results from both experiments are summarized in Table 4.1 where the VMA-based page counts are reported in brackets.

	Reported by	**Pages**	**VMAs**	**Pointers**
Apache	Our Tool	1021	53	23406
	Kernel	1033 (1470)	37	N/A
Ustealer	Our Tool	884	26	9544
	Kernel	909 (4928)	13	N/A

Table 4.1: Apache and Ustealer's address spaces

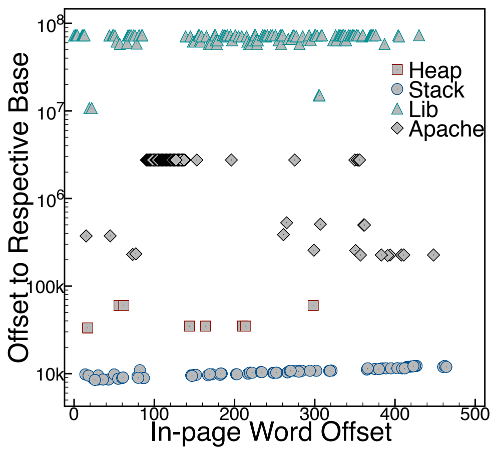
Our tool identifies most pages which do have mappings in the page tables, and more VMAs than reported by the kernel due to the kernel's lazy memory allocation. For the sake of better performance, the kernel may allocate the virtual address space without mapping *all* physical pages, which lead to discontinued VMAs. Our

[1]https://github.com/atmoner/Ustealer

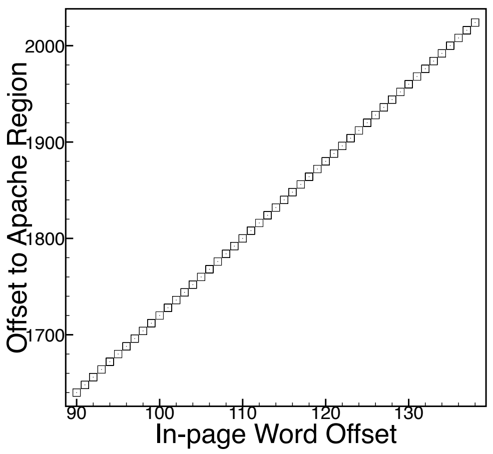
pointer-test results hence reflect the actual mappings defined in the page tables, which are more fine-grained and more accurate. Note that in the experiment with Ustealer, the kernel allocated 4928 pages while only 909 pages are actually mapped and only 13 VMA regions are reported. This indicates that the malware aggressively requests the kernel to allocate large chunks of memory. Our measurement shows that the page fault handler execution takes about 194 nanoseconds. It takes the tool about 26 milliseconds and 40 milliseconds to explore Ustealer and Apache, respectively. The main overhead is due to page faults induced by non-pointers in the pages.

We run an analysis upon the 287 (potential) pointers discovered on one 4 KB stack page of Apache which accommodates up to 512 8-byte pointers. According to the pointer values, we group them into four categories: heap pointers (i.e., those pointing to the heap object), stack pointers, library pointers and Apache pointers (i.e., those pointing to Apache's code and data sections). A graphic representation of them is in Figure 4.4(a) which shows a seemingly flat line formed by Apache pointers. Figure 4.4(b) visualizes that segment and shows that they are 49 adjacent pointers pointing to adjacent memory locations in Apache's data section. With a high probability, they are an array of pointers pointing to another array.

Our analysis tool also attempts to discover possible link-lists using a chain of pointers in the heap. The analysis is based on the pattern that pointers in link-list nodes always have the same offset to the respective node' address. The results are reported in Table 4.2, where the third column is the distance between the pointer's address and its node address, and the last column is the number of pointers pointing to the head of the list. Interestingly, there is one list whose node structure has the pointer as the first member.



(a) Visualization of pointers pointing to four different VA regions in Apache



(b) Visualization of 49 adjacent pointers in Apache

Figure 4.4: Visualization of pointers discovered on a stack page. X-axis is the offset of the pointer's location in the page in terms of the number of 8-byte memory words; Y-axis is the offset of the pointer's value with respect to the base address of the corresponding zone (displayed in logarithm scale in Figure 4.4(a) and in linear scale in Figure 4.4(b)).

Head Node Addr.	**# of nodes**	**Offset**	**# of References**
0x5555557FC260	23	32	5
0x5555558007F0	22	24	7
0x555555800110	10	0	1

Table 4.2: Potential link-lists in one of Apache's heap pages

4.3 Non-Intrusive System Call Monitoring

This tool captures a user-mode program's system call issuance without intercepting its execution. In this case, we do not consider the scenario where the target kernel intentionally hides its user program's system call events by not respecting the system call convention.

4.3.1 Design Sketch

The evolution of the kernel stack during a system call is depicted in Figure 4.5. Whenever the target process runs in user-mode, the process's kernel stack is empty. In response to the `syscall` instruction, the kernel always first pushes the user-mode stack segment and stack pointer into the kernel stack, followed by other information including the system call number, the return address and parameters. When the system call handler completes its service, it pushes `RAX` storing the system call return value into the stack, before firing the `sysret` instruction. In the end, the kernel stack pointer is restored to its initial position. Note that the contents of the stack are *not* zeroed.

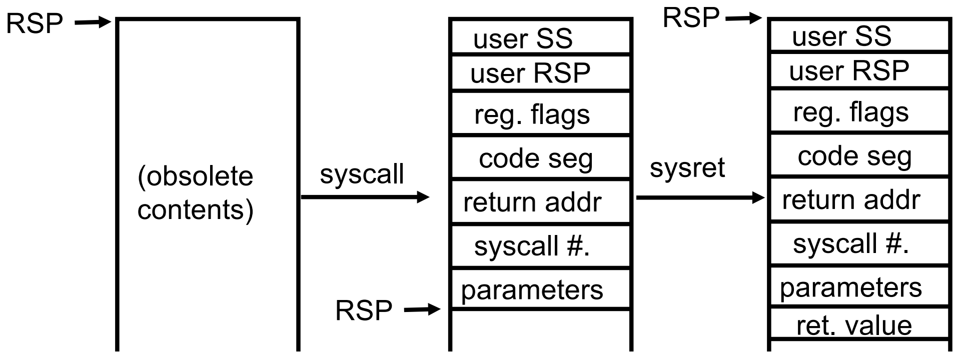


Figure 4.5: Evolution of kernel stack from receiving syscall to completion of sysret

The tool continuously monitors several kernel stack placeholders which are possibly updated due to a system call, including the user-mode stack pointer, the return address, the system call number and parameter(s). If the contents in those placeholders are found to be different from those recorded in the prior system call, the tool reports a new system call issuance and saves the user-mode `RSP`, `RIP`, `RAX` images, six syscall arguments, the syscall return value, and RFLAGS. By monitoring a combination of placeholders, we maximize the likelihood to catch the event. It is possible that repeated system calls do not incur any change in those monitored stack placeholders and therefore are missed by our tool. For instance, the subject is a network daemon keeps polling the socket while no packet arrives.

4.3.2 Evaluation

Our implementation consists of 121 SLOC. It takes merely around 20 CPU cycles to record one system call, which is speedy enough to ensure that the tool can resume its monitoring before the kernel handler returns. Since the system call issuance is a blocking operation, the application cannot issue another system call during kernel service. We have tested our tool against four unnamed user space malware samples downloaded from Github [81] and two benign programs (Firefox and Bash). In the experiment, we stop malware execution after catching 100 events, and use system calls recorded by `strace` in the guest as the baseline to evaluate monitoring effectiveness. The results are shown in Table 4.3 where M1 to M4 denote four unnamed malware samples.

Our tool captures all system calls from Firefox and Bash, but misses a significant percentage of system calls from M2 and M4. After manual investigation, we find that both malware continuously issue system calls from the same calling site which does not change the user-stack pointer stored in the kernel stack.

Target	M1	M2	M3	M4	Firefox	Bash
# Captured	100	100	100	76	84	57
# Noise	0	0	0	24	16	43
# Missed	0	39	0	308	0	0

Table 4.3: Results of system call monitoring

Comparison With Existing Methods. The main advantage of onsite-based system call monitoring is its non-intrusiveness. No hook is placed into the subject or the guest kernel. Neither the target execution flow is intercepted. It is therefore more convenient to deploy and more difficult to detect,

The main disadvantage of onsite-based system call monitoring is the “noise”, since exceptions and interrupts are also caught because they induce the kernel stack modification as well. Differentiating system calls from interrupts and exceptions requires a deeper analysis on the stack. One indicator is the `RF` bit (i.e. the 16th bit) in RFLAGS register image stored in the stack. For any fault-class exceptions, the hardware sets this bit in the register’s stack image; otherwise it is cleared. Therefore, it is an accurate indicator to differentiate fault-class exceptions and system calls. Nonetheless, an analyzer may benefit from these noises because they provide extra intelligence to understand the subject behavior. For instance, frequent non-present page faults indicate the subject’s intensive memory usage and frequent interrupts may indicate intensive I/O operations.

4.4 Combining with Static Analysis Tool

OASIS not only facilitates the onsite analysis on a program’s runtime memory, but also can combine with existing static analysis tool. Static analysis tool normally takes a ELF/PE format object file as input, while some applications is likely to emit informative binaries at runtime which needs analysis. For example, JavaScript(JS) interpretively executes in a confined environment which constrains its malicious behaviors, so that the runtime emitted JIT code becomes the popular target for a JS malware. Memory overflow vulnerabilities are here and there, which leaves chance for a JS malware to trick JIT compiler to emit arbitrary JIT code. Therefore, analysis on these runtime generated JIT code is necessary. In the following, we show that an existing static binary analysis tool can run on onsite mode which takes the JIT code from memory as input directly and generate a Control Flow Graph(CFG) for it.

4.4.1 Design Sketch

Dyninst[2] allows CFG generation on ELF/PE format object file among its strong binary analysis capabilities. In order to reuse their ParseAPI to analyze on snippets of JIT code in memory directly, we implement a new code source that can represents binary from memory.

Dyninst recursively disassembles the binary, which means the target of all `jmp/call` instruction would be remembered and used as the start point of next parsing. It is worth to note that all the pointers from the target program reserve their semantics under onsite mode. Therefore no additional mapping information is needed for Dyninst.

The JS engine needs to share the base address and size of the JIT code with Dyninst whenever there is one emitted. They first negotiate a page, and then share all the necessary information there, including a flag to indicate there is a new JIT code emitted, and the base address and size of the new emitted JIT code. Since CFG tool running in onsite mode has read access to the target program's address space, it can continuously checks whether the flag is set.

4.4.2 Attacks on intermediate data of JIT compiler

The JIT compiler is responsible for generating native code for functions and methods that are invoked frequently. There are several steps that a JIT compiler takes to convert bytecode into native code. An Intermediate Representation(IR) is first generated, then it is optimized, register allocated, etc. Once the IR is ready, it will be encoded into native code. Most of the JS engine would put the finalized JIT code into a readable-and-executable region to mitigate attacks which overwrites the JIT code directly. However, since the IR must store in a writable memory region, the attackers can trick the JIT compiler to generate arbitrary malicious payloads by manipulating the IR before they are used as inputs to the JIT compiler [82].

[2]http://www.dyninst.org/

For ChakraCore [83], which is the JavaScript engine of Internet Explore, it allocates a buffer which temporarily holds the native code. After all of the native instructions have been emitted into the buffer, the JIT compiler relocates the buffer into a readable-and-executable region. Theori [84] targeted on this temporal buffer to overwrite the native instructions there directly. We emulate this attack on Chakaracore 1.4.0. Since the modification is conducted after compilation, there is no check by the compiler to discover the attack. any payload is reasonable to bypass. In order to mitigate this attack, Microsoft patched the JIT compiler with a cyclic redundancy checksum of the emitted instructions during compilation, the JIT code is only executed if the checksum of the relocated buffer corresponds to the original checksum. In order to emulate this attack, we modify ChakraCore to let this checking pass at any condition. The payload we emulated is same as [84] which contains two stage. The native code in the temporal buffer is overwritten with the first stage payload which issues a `mprotect` syscall to map a memory region as RWX, and then jumps there. The second stage payload can be large enough to contain arbitrary code.

4.4.3 Implementation

We places hook inside ChakraCore, so that its JIT compiler would share the start address and size of the native code whenever there is one emitted. For more specifically, the hook is placed inside `Encode::Encode()` which converts IR into native code. Every JIT code is associated with a `JITOutput` instance. We get the start address of the JIT code and its size through class `JITOutput`'s member methods `GetCodeAddress()` and `GetCodeSize()` respectively.

In Dyninst, the `CodeSource` interface is used by the `ParseAPI` to retrieve binary code from an binary code object. The `ParseAPI` provides a default implementation based on the `SymtabAPI` that supports many common binary formats including ELF, COFF, and PE. We implement a set of member methods in class `CodeSource` to support parsing binary code in memory. After constructing a `CodeSource` instance for the new generated JIT code, `parse()` function is invoked to analyze the binary. Whenever there is a new JIT code emitted, a new `CodeRegion` is added into the existing `CodeSource` instance. `parse(CodeRegion*, Address, bool recursive)` function can be invoked repeatedly to analyze on the newly added JIT code.

4.4.4 Experiments

We test on a benchmark JS from JetStream[3]. As shown in Table 4.4, there are 20 JIT code emitted sequentially with average size of 0x686 bytes at runtime. The CFG generation tool is able to catch up the emitting of all the 20 JIT code and parsing them. It takes 0.767ms to parse each JIT code in average.

# of JIT code	Average Size	Average Time	# of BasicBlocks	# of Edges
20	0x686	0.767	1665	2290

Table 4.4: Performance of CFG generation tool

Among these 20 JIT code, we rely on gdb to dump one from runtime memory with size 0x128. We then rely on *objdump* with `-b binary` option to disassemble it, a manual comparison shows the result matches with the one generated by CFG tool in onsite mode. We also run the CFG generation tool on the emulated attack, it successfully identifies the first stage payload. While since the first stage payload use an indirect `jmp` to jump to its second stage payload, the tool lacks the ability to identify the target for an indirect jump.

4.5 Summary

In this chapter, we developed several exemplary OMA tools. The first one demonstrates the native access speed guaranteed by OASIS. The speed facilitates the memory evolution monitoring, which further helps to infer the program behaviors such

[3]http://browserbench.org/JetStream/in-depth.html#hash-map

as the system call events as shown in the third tool. Without memory dumping or reliance on the target kernel objects such as VMA structures, the second analyzer efficiently plots the running target's virtual address space layout and discovers pointers and link-lists via conducting large-scale and on-demand virtual address testing. The last one demonstrates that existing static analysis tools can be adapted for onsite analysis.

Chapter 5

OASIS based Execution Flow Instrumentation

In this chapter, we propose a new dynamic analysis model named **Execution Flow Instrumentation(EFI)**. The concept is for a user-space program (denoted as the *analyzer*) to *instrument the execution flow of a malicious thread across user and kernel spaces*. Essentially, the target's and the analyzer's instruction streams are interlaced at junctures chosen by the latter. By fusing their execution flows instead of the static code, we aim to combine the virtues of hardware-trapping (strong security and non-intrusiveness) and code instrumentation (native-access) without their drawbacks.

How to securely and efficiently interleave the analyzer instruction flow with the target flow in an on-demand fashion? It is not secure to mix up their instructions in one address space as in code instrumentation. The hardware event trapping approach follows the ideology of isolation rather than integration, as it leads to executions in a more privileged setting and also has other drawbacks such as non-negligible trapping overheads and limited triggering conditions provided by the hardware.

OASIS provides the onsite environment where the analyzer can natively read-/write into the target's virtual memory. To realize the EFI, we extend OASIS with another address space hierarchy in onsite environment, where the target runs with all physical pages in the guest and all access permissions set as in the guest. We also extend OASIS with a significant instrument technique whereby the analyzer securely and flexibly chooses the junctures to regain the execution flow. To the analyzer, this is done as native as in the code instrumentation including the memory object and register information introspection, while the instrumentation remains transparent and inevitable as in hardware trapping based schemes. The transitions do not entail any CPU privilege/mode changes. These advantages are derived from one feature of onsite environment: the target's address space is shared with the analyzer, but not vice versa. In contrast, they share no address space in event-trapping systems [6, 7, 1, 2] and share the space entirely and mutually in code-instrumentation systems [16, 4].

We developed EFI tools in three case studies. The first case study uses an EFI tool to trace full-space control flow transfers including asynchronous executions such as page fault handling. The second case study consists of two EFI tools working in tandem with Syzkaller [19], a popular kernel fuzzer from Google. One tool is for postmortem analysis based on a kernel crash report. It uses instruction slicing to identify instructions relevant to a data flow and then traces their execution. The other EFI tool analyzes the Syzkaller executor thread exploring a malicious kernel driver. It uses both breakpoints and tracing to uncover the driver behavior which successfully evades the fuzzer's logging mechanism. The third case study examines how a user-space program exploits the vulnerability in dynamically loaded kernel modules. All EFI tools are developed and launched as applications despite making kernel analysis. The case studies show that EFI tools are handy and easy-to-develop. They are especially well-suited to fine-grained and agile analysis on (malicious) kernel threads.

Organization. Next, we present an overview of OASIS based EFI and the workflow of dynamic analysis using EFI . Analyzer's execution in onsite environment is discussed in 5.2. The details of target execution in onsite environment is presented

5.3. EFI techniques are elaborated in 5.4 followed by its security and transparency analysis in 5.5. The case studies are presented in 5.6. The implementation details of the case study tools are in 5.7.

5.1 Overview

5.1.1 Overview of OASIS Based EFI

We extend OASIS to support EFI with its new architecture shown in Figure 5.1. It has a new component OASIS-Lib which consists of one code page and several data pages to facilitate 1) control transfers between the target and the analyzer; and 2) event handling during analyzer execution. The Trampoline is used to handle the analyzer's interactions with the host OS to provide transparent OS services to the analyzer which has been discussed thoroughly in Chapter 3. The main extension is on the Manager which sets up and manages the *onsite environment* for both the analyzer(including OASIS-Lib) and the target's execution.

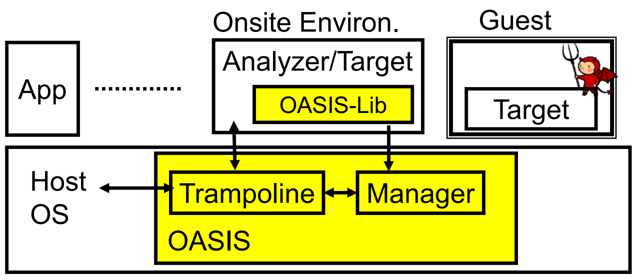


Figure 5.1: The architectural view of OASIS based EFI. The entire guest is untrusted including the guest kernel. The boxes in yellow illustrate OASIS software composition.

The onsite environment consists of one vCPU, two suites of EPTs (denoted as *A-EPT* and *T-EPT*), and four paging structure pages (denote as O-PML4, O-PDPT, O-PD and O-PT). The analyzer is loaded into the environment immediately after process creation, while the target thread in the guest is captured and migrated to it.

- When the analyzer runs, the onsite environment uses A-EPT and O-PML4 to instantiate the *analyzer-target paging hierarchy* that merges the target space with the analyzer.

- When the target thread runs, the onsite environment uses T-EPT, O-PML4, O-PDPT, O-PD and O-PT to instantiate the *target-lib paging hierarchy* that merges the target space with OASIS-Lib. All the target's memory accesses are still within the guest.

Our approach to analyzer-target execution interleaving is to switch the underlying EPT paging hierarchy without triggering an interrupt or exception. The switches are realized through two short instruction segments in OASIS-Lib. The one for switching from the target to the analyzer is the *exit-gate*, and the other is the *entry-gate*. A combination of techniques are used to safeguard their security and transparency.

5.1.2 Workflow of Dynamic Analysis using EFI

The high level workflow of dynamic analysis using EFI is illustrated in Figure 5.2. To start the analysis, OASIS exports the target threads from its core in the guest to the onsite core. The analyzer's EFI session consists of multiple rounds of target-analyzer execution flow interleaving at junctures chosen by the analyzer. If needed, the analyzer restores the target back to the guest to continue its execution. The cycle repeats until the analyzer completes the entire analysis task.

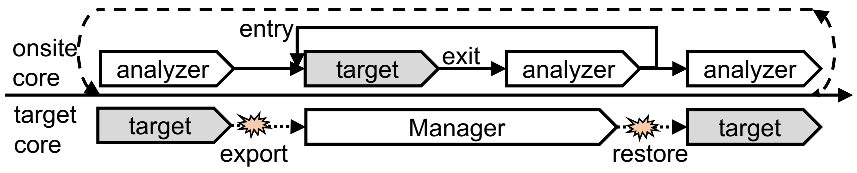
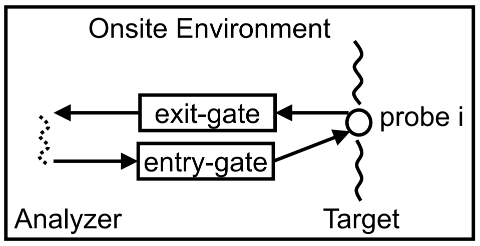
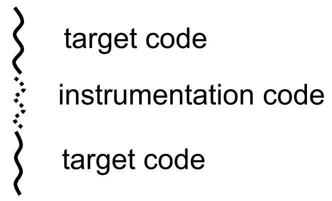


Figure 5.2: The high-level workflow of dynamic analysis using execution flow instrumentation. It consists of cyclic switches between the target and the analyzer within the onsite environment.

The analyzer uses *probes* to specify the junctures for instrumentation. When the target execution flow reaches it, the probe transfers to the exit-gate which further jumps to the analyzer code. After the analyzer completes execution of the instrumentation logic, it jumps to the entry-gate which further returns the control back to the target. Although the use of probes changes the target code running in the onsite environment, it neither alters the target address space or affects other threads sharing the code in the guest. OASIS ensures its transparency by setting it as execution-only. Figure 5.3(a) depicts the i-th round interleaving in an EFI session which has the same effect as code instrumentation in Figure 5.3(b), but with stronger security and transparency.



(a) EFI across address spaces with no CPU privilege or mode changes



(b) Equivalent effect by using code instrumentation

Figure 5.3: Illustration of execution flow instrumentation via probes and gates.

5.2 Analyzer Execution in Onsite Environment

Recall that the system service requests from the analyzer in onsite environment are transparently handled by the host OS. We briefly review its address mappings here. When the analyzer runs the underlying EPT is A-EPT. All entries on the target's PML4 are copied to O-PML4. The guest's GPA domain is denoted as $[0, T]$, the `CR3` on the onsite core is loaded with GPA $\alpha \in_R [T, 2^{48}]$, namely α is randomly chosen between T and 2^{48}. Among the kernel half of the VA domain, we choose an unoccupied 512-GB band for the analyzer. Without loss of generality, suppose that the λ-th 512-GB band is chosen, where $256 \leq \lambda < 511$. The λ-th entry in O-PML4 is populated with a randomly generated GPA $\beta \in_R [T, 2^{48}]$ with the supervisor bit cleared. A-EPT is configured as follows to map GPAs to HPAs.

- For all GPAs used in the guest, their GPA-to-HPA mappings are the same as in the guest and the permissions are set as non-executable.
- GPA α is mapped to the HPA of O-PML4 and GPA β is mapped to the HPA of the analyzer's PDPT page in the host.
- For all HPAs appearing in the analyzer's PDPT, PDs, and PTs, their GPAs are set equal to themselves. Namely, A-EPT uses the identity map for all of them.

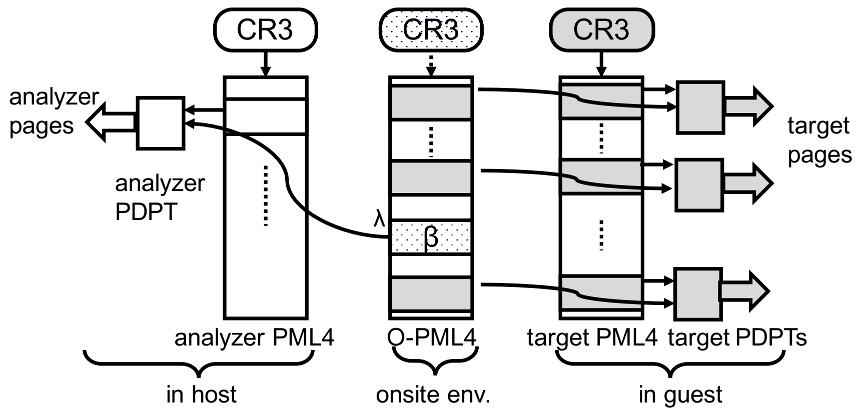


Figure 5.4: The analyzer-target hierarchy is rooted at O-PML4. The boxes with patterns indicate randomized values. All arrows except those in the host involve the underlying GPA-to-HPA mapping in A-EPT.

The resulting analyzer-target hierarchy is shown in Figure 5.4. All target VAs are mapped in the same way (except permissions) to the corresponding physical pages as in the guest. Hence, the analyzer instructions can use those VAs to natively read/write accesses to the target data and code pages. All analyzer VAs are also mapped to the corresponding physical pages since the MMU travels the same PDPT page and subsequent structures as in the host.

Runtime Update. The guest kernel's updates on the target's paging hierarchy (except its PML4) and the host OS's updates on the analyzer's paging hierarchy are both automatically take effects in the onsite environment since their PDPTs, PDs, and PTs are linked to O-PML4. To ensure consistency between the target's PML4 and O-PML4, OASIS sets the former as read-only by configuring the guest

EPT. The guest kernel's PML4 update is intercepted and cloned to O-PML4, which occurs seldom in 64-bit platforms.

Analyzer Execution. Initially, the analyzer runs in Ring 3. It may also run in Ring 0 by promoting itself for privileged operations (e.g., accessing the target kernel) or by following the target kernel-mode execution through the exit-gate. In order to support self privilege escalation and to handle exceptions, OASIS provides new IDT, GDT and TSS. The GDT has a call gate for the analyzer own privilege escalation. Note that these descriptor tables are exclusively used for the analyzer's execution, they are different from the ones when the target runs.

5.3 Target Execution in Onsite Environment

The target thread is exported to the onsite environment. We describe below its underlying target-lib hierarchy, consistent execution, and target-analyzer control transfers.

5.3.1 Address Mapping For Target Execution

Constructed by using T-EPT, the target-lib hierarchy merges the target address space and OASIS-Lib's. Its details are elaborated in Figure 5.5. Both `CR3` and O-PML4 remain the same as in the analyzer-target hierarchy. T-EPT clones all GPA-to-HPA mappings in the guest *including their permissions*. To prevent the target from detecting and accessing OASIS-Lib, we randomize both its base VA within the 512 GB band occupied by the analyzer and also the GPAs of the paging structure pages used in translation. Specifically, T-EPT maps GPA β to O-PDPT's physical address. The GPAs for O-PD and O-PT are also randomly selected from range $[T, 2^{48}]$. According to the random VA assigned to OASIS-Lib, the corresponding entries in O-PDPT, O-PD, and O-PT are assigned properly to construct traversal paths leading to OASIS-Lib's physical pages. Note that O-PDPT, O-PD and O-PT are exclusively for OASIS-Lib, no other entries on them are populated.

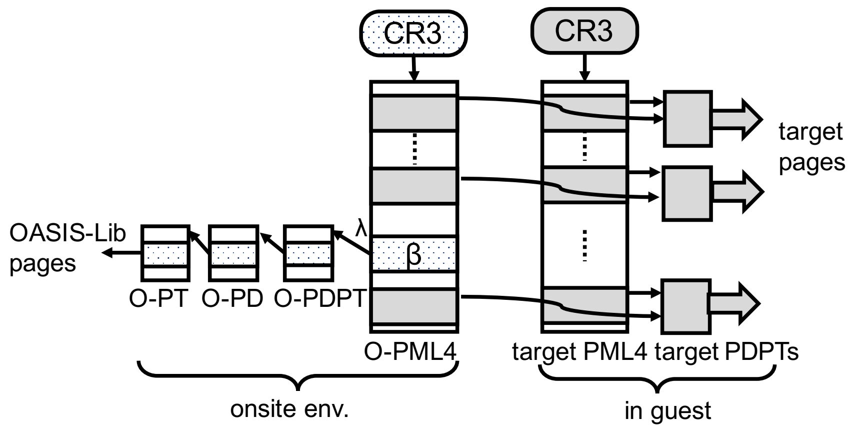


Figure 5.5: The target-lib hierarchy rooted at O-PML4. The boxes with patterns have randomly assigned GPA contents. All arrows use the underlying GPA-to-HPA mappings in T-EPT.

5.3.2 Execution Consistency

With the target-lib hierarchy, the exported target has consistent memory references. Hence, the remaining issues are to handle the CPU context as well as system-level structures that are vital for interrupts and exceptions.

Exportation & Restoration. The target thread is captured via VM-exit in the guest so that its entire CPU context is saved to the main memory. The context includes general-purpose registers, control registers and model specific registers (MSRs). To export it to the onsite environment, OASIS configures the VMCS of the onsite core according to the saved target core context, except that `CR3`, `CR4`, `IDTR`,`GDTR` and `TR` are the same as in analyzer execution.

The trapped target core is hold by OASIS until the target thread is restored, to mimimic the target's supposed CPU occupancy in the guest and also facilitate subsequent restoration and I/O operations. Upon the analyzer's request to restore the target, OASIS updates the target core VMCS structure with the onsite core's (including `RIP`) so that the target continues its execution in the guest from the context in the onsite environment.

I/O Operations.

In the onsite environment, the target directly accesses the guest's memory-mapped I/O regions and DMA buffers via their VAs. However, port I/O operations and interrupt delivery do not use virtual addresses and hence require special treatments. The design is dependent on the underly I/O mechanism provided by the host OS to the guest.

In Linux KVM, I/O requests are trapped to the hypervisor which dispatches it to QEMU to execute. When the hardware completes the task, the external interrupt is therefore delivered to QEMU which notifies the hypervisor to inject the interrupt into vCPU during VM-enter. In OASIS, the general idea is to let the Manager to replace QEMU. The target's I/O operation in the onsite environment is trapped to the Trampoline which passes it to the Manager withholding the target core in the guest (shown in Figure 5.2). The Manager executes the operation so that it appears to the host OS as a request from the target core. After I/O operation completion, the target core's VM re-entry is intercepted by the Manager which then notifies the Trampoline in the onsite core to resume the target execution and inject the external interrupt if any. In this way, the exported target has the same behavior in I/O operations and interrupt delivery as in the guest.

System Data Structure Relocation. To support analysis on exception and interrupt handling in the target thread, OASIS relocates the target's system-level data structures to OASIS-Lib, including the IDT, GDT, TSS. For TSS relocation, it also updates the TSS descriptor in the GDT accordingly. The target's exception and interrupt handling are the same as under the original setting.

To protect transparency, OASIS prevents the target from accessing those registers by configuring the VMCS structure. Any software access to them is trapped to the analyzer which returns the original address. Note that the target thread's read to these tables using their original VAs are not affected, because thee tables are physically in the guest and are not changed. Updates to the tables are intercepted so that OASIS clones the changes to the relocated counterparts.

The relocation allows the analyzer to customize these data structures in order to monitor and control asynchronous events in target execution. For instance as described later in 5.4, an analyzer interested in the target's page fault handling may hook the target's #PF hander to capture the event and then traces the execution. Note that the analyzer has its own set of system-level data structures which are provided by and located in the host kernel.

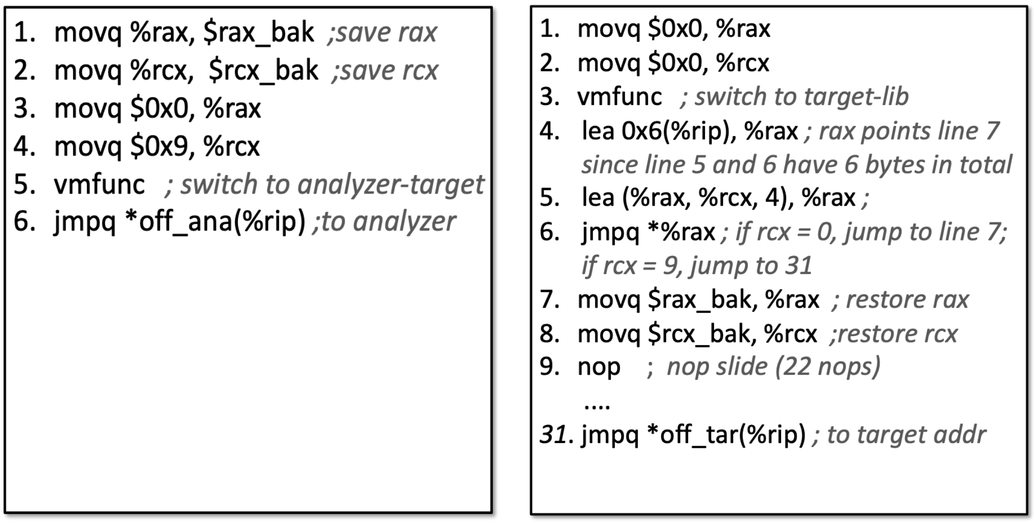
5.3.3 Cross-Flow Control Transfer

The target execution in the onsite environment is instrumented via cross-flow control transfers. The basic idea is to switch the EPTs by using the `vmfunc` instruction[1] in order to switch the paging hierarchies for the execution flows. After the switch, the CPU fetches the next instruction from the new address space as it translates VAs under the new hierarchy.

Straddling in both hybrid hierarchies, the exit-gate switches from the target-lib hierarchy to the analyzer-target hierarchy while the entry-gate switches in the opposite direction. The two gates are in the OASIS-Lib code page which is set as writable in the analyzer-target hierarchy in order for the analyzer to flexibly customize the entry-gate. An OASIS-Lib data page is used to save registers and to facilitate control transferring to destinations more than two GB away from the gates.

Exit-gate. Figure 5.6(a) presents the assembly code of the exit-gate which is called from the target flow to pass the control to the analyzer flow. It first saves the target's current `RAX` and `RCX` to the pre-defined locations in the OASIS-Lib data page as the two registers are needed to load `vmfunc` parameters. A-EPT is pointed by the 9th EPTP in the pre-prepared EPTP-list. It issues `vmfunc` with index 9 specified in `RCX` to instruct the hardware to switch to A-EPT. Finally, it jumps to the analyzer's handler with an indirect IP-relative `JMP` which fetches the destination from OASIS-Lib's read-only data page. Note that instruction at Line 6 is fetched and executed from the analyzer-target hierarchy. The hierarchy switch does not disrupt the instruction sequence because the OASIS-Lib page containing

[1]According to Intel specification, `vmfunc` invokes a predefined hypervisor function specified by `RAX` and `RCX` without incurring VM-exit.



```
1.  movq %rax, $rax_bak  ;save rax
2.  movq %rcx,  $rcx_bak  ;save rcx
3.  movq $0x0, %rax
4.  movq $0x9, %rcx
5.  vmfunc   ; switch to analyzer-target
6.  jmpq *off_ana(%rip) ;to analyzer
```

(a) Exit-gate

```
1.  movq $0x0, %rax
2.  movq $0x0, %rcx
3.  vmfunc   ; switch to target-lib
4.  lea 0x6(%rip), %rax ; rax points line 7
    since line 5 and 6 have 6 bytes in total
5.  lea (%rax, %rcx, 4), %rax ;
6.  jmpq *%rax ; if rcx = 0, jump to line 7;
    if rcx = 9, jump to 31
7.  movq $rax_bak, %rax  ; restore rax
8.  movq $rcx_bak, %rcx  ;restore rcx
9.  nop    ;  nop slide (22 nops)
    ....
31. jmpq *off_tar(%rip) ; to target addr
```

(b) Entry-gate

Figure 5.6: Pseudo-assembly code of the exit-gate that passes the control to the analyzer and the entry-gate that returns the control to the target.

the gate is mapped as executable in both hierarchies with the same VA and PA. To minimize the exit-gate's code size, it does not save the target's CPU context except two registered used by itself. It is the analyzer handler's responsibility to retrieve the target's RAX and RCX from the data page and save the target CPU context.

Entry-gate. Figure 5.6(b) presents the assembly code of the entry-gate which is called from the analyzer flow. It issues vmfunc with index 0 specified in RCX instructing the hardware to switch to T-EPT. Line 4 to 6 check the value of RCX in case the target jumps to Line 3 directly with RCX prepared by itself. From line 7, it restores RAX and RCX and jumps to the destination specified by the analyzer. None of the instructions before line 7 modifies the CPU context except RAX and RCX. Especially, RFLAGS are not affected. The transfer destination in line 31 is also loaded from the read-only data page with an indirect IP-relative JMP. Note that it is the *analyzer*'s responsibility to prepare the desirable CPU context (including the transfer destination) for the target to resume its execution. The instructions following vmfunc are fetched from the target-lib hierarchy. A slide of 22 nop instructions is placed before the final jmp instruction. The slide is long enough for two instructions used for make-up execution due to the probe (as explained in 5.4.2). We discuss the security of the gates in 5.5.1.

In short, the cross-flow control transfers do not incur any CPU privilege or mode changes. As shown in Figure 5.7, OASIS-Lib is mapped into the same VA in both hierarchies. The analyzer and the target use the same set of `IDTR`,`GDTR` and `TR` while their hierarchies map them into different sets of physical pages. Hence, the analyzer does not need to set those registers in a cross-flow switches, which may demand the analyzer to escalate its privilege.

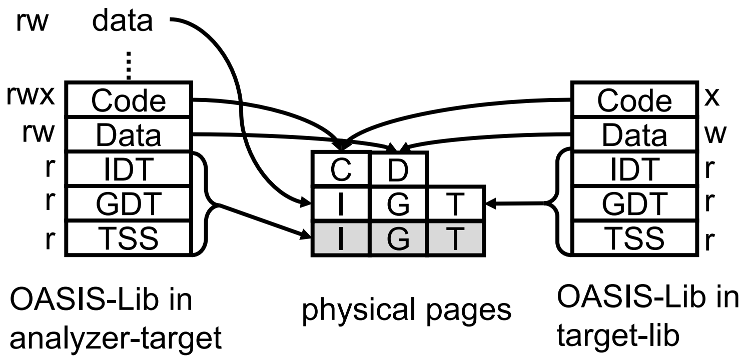


Figure 5.7: The OASIS-Lib are mapped in the same VA region in both hierarchies. The shadowed physical pages are for analyzer execution only. The analyzer accesses the target's IDT, GDT and TSS from its data section VAs.

5.4 Execution Flow Instrumentation

Recall that execution flow instrumentation (EFI) interlaces the analyzer's execution into the target flow at chosen junctures and that the exit/entry-gates are the switches between the two executions flows. In this section, we explain how the analyzer flexibly and securely specifies EFI junctures. The basic idea is to use *probes* [17, 28] to replace target instructions at the desirable virtual addresses. When the target flow reaches a probe, the flow is diverted to the exit-gate and further to the analyzer.

5.4.1 Page Substitution for Probe Installation

Probe installation inevitably changes the target code, though not the virtual address space. To support secure and transparent probe installation, OASIS offers the *page-substitution* and *page-reinstatement* hypercalls for the analyzer to substitute a target physical page with a new physical page from the host memory. The mechanism is illustrated in Figure 5.8.

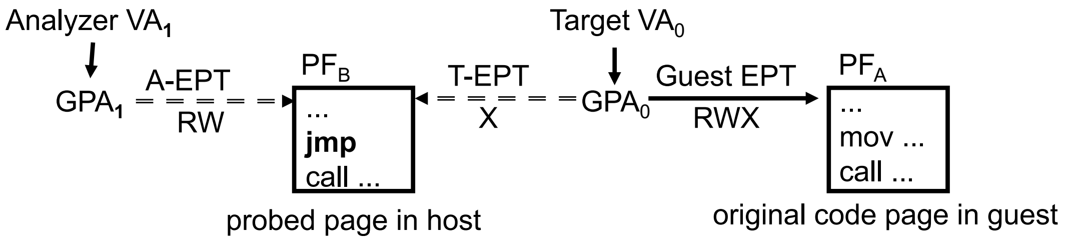


Figure 5.8: EPT redirection is applied on T-EPT for probe installation and uninstallation.

Suppose that target code page VA_0 is mapped to physical page frame PF_A in the guest via the guest physical address GPA_0, and that the analyzer has a data page VA_1 mapped to physical page frame PF_B in the host. To install a probe in VA_0, the analyzer copies it to VA_1, modifies VA_1 with the probe, and then issues the page-substitution hypercall. In response, OASIS modifies the GPA-to-PA mapping in T-EPT to map GPA_0 to PF_B with execution-only permission to prevent the target from reading/writing it. As a result, when the target control flow reaches VA_0, it fetches instructions from PF_B instead of PF_A. The analyzer running under A-EPT can read/write the probed page PF_B via VA_1. When the probe in VA_0 is no longer visited, the analyzer uses the page-reinstatement hypercall to restore the original entry in T-EPT so that it is mapped to PF_A.

Permission Conflict Resolving. For the sake of transparency, we follow the approach in SPIDER [7] to redirect the target's read and write to VA_0 back to PF_A. The basic idea is to load the onsite core's data TLB with EPT mappings to the original code page (with read-only permission) and to load the instruction TLB with EPT mappings to the probed page (with execution-only permissions). The target's write to VA_0 is single-stepped so that a modification on PF_A is also cloned to PF_B. The key difference between OASIS and SPIDER [7] is that the data TLB loading is through OASIS-Lib instead of the target's own instructions. Therefore, there is no need to single-step reading instructions in OASIS.

Mapping Modification. The exported target may change its VA-to-GPA mappings. It is likely that VA_0 is re-mapped to GPA_0^* either out of benign reasons such as page swapping or for a malicious purpose. Although the change does not affect the target execution in the onsite environment, it invalidates the probe in VA_0 since PF_B will not be accessed any more when executing code in VA_0. To resolve the issue, OASIS traps such a modification by configuring T-EPT to write-protect the paging structure pages translating VA_0. If the trapped modification maps VA_0 to GPA_0^*, it updates T-EPT to map GPA_0^* to PF_B. Note that only the mappings to the probed pages are under monitoring.

It is possible that another kernel thread in the guest updates VA_0's mapping in parallel. Such an update is not subject to T-EPT restrictions and therefore is not trapped. However, since the thread runs in a different core from the target core, it is expected to notify the target thread to invalidate the TLB at the target core so as to avoid mapping inconsistence. The cross-core notification is captured by OASIS which then updates T-EPT accordingly.

5.4.2 EFI Probes

An EFI probe kickstarts the transition from the target flow to the analyzer. Depending whether it is removed after being triggered or not, a probe can be used for tracing or as a breakpoint. We propose two EFI probes: INT3-probe and JMP-probe for the analyzer to use, although the analyzer can use its own probes. The INT3-probe, as in [17, 7], is a one-byte instruction (opcode `0xCC`) and can be used for tracing or as a breakpoint. When the probe triggers the INT#3, the exception handler jumps to the exit-gate. (Recall that the target's IDT and GDT are relocated and modified to install new handlers provided by the analyzer.)

The JMP-probe jumps to the exit-gate without triggering any hardware event. The main challenge stems from the addressing modes in x86-64. In the address space mapped by the target-lib hierarchy, the distance between the exit-gate and the target code page containing the probe can be more than 2GB, which is beyond the reachable range of any addressing mode if no general register is used. However, a register indirect transfer requires the probe to save the register to the memory, which is intrusive and undermines transparency.

In OASIS, the JMP-probe is a far jump instruction to a call gate which transfers the control to the exit-gate. The probe is in the form of:

```
REX.W ljmp *offset(%rip)
```

There are three micro-operations in a `ljmp`. 1) The CPU fetches 80-bit from `offset(%rip)` and takes the upper 16-bit as a selector; 2) The CPU gets the call-gate entry from the GDT/LDT based on the selector; 3) The CPU uses the `address` from the call gate entry directly as the control transfer destination if the permission checking passed. The `ljmp` transfers to the exit_gate because its address is specified in the `address` field of the call gate entry. The call gate's Descriptor Privilege Level(DPL) is set as 3 to allow the invocation from both user and kernel privilege, while the Code Segment(CS) selector in the call-gate entry is set as Ring

0 or Ring 3 selector to transfer to the kernel or user privilege respectively.

When installing the JMP-probe, the analyzer constructs the probe instruction on-the-fly by determining the value of `offset` so that (1) the instruction operand is a far pointer pointing to the desired call-gate selector; and (2) the referenced memory page is read-only so that the selector is not modified after probe installation.

The main steps for composing the probe is as follows. Initially, the analyzer populates all unused GDT entries with the call gates. To install a JMP-probe during an EFI session, the analyzer randomly picks 16 bits from one target code page with the only restriction that the 3rd least significant bit be '0', so that they form a GDT segment selector. It then checks whether bits 3 to 15 form up a valid a call gate index. Since there are up to 8K valid GDT indexes, the checking succeeds with a almost 1 probability[2]. If it fails, the analyzer picks and checks another 16 bits. Once a selector is found, the analyzer calculates `offset` according to its virtual address. Note that a selector can be used for any JMP-probe installation within ± 2 GB range. Hence, the analyzer does not have to search it for each probe installation. Since the JMP-probe is multiple bytes long, it cannot be used as a breakpoint due to unintended transfers and attacks on transparency.

Makeup Execution

A thorny issue about probes is the makeup execution of the target instruction(s) being replaced. Two solutions are used in the literature. One is to emulate those instructions in a separate space and the other is to restore them and single-step them before replacing them with the probe again [7]. Obviously, both solutions are cumbersome and incur significant CPU time.

Attributing to the analyzer's native-access to the target, it is comparatively easier to resolve the issue in OASIS. For the JMP-probe, the analyzer writes back the replaced target instruction(s) and resumes the target execution from the probed VA.

[2]Since most Linux kernel uses less than eight entries in the GDT, the success probability is $\frac{2^{13}-8}{2^{13}} \approx 0.999$.

For a transfer instruction affected by the INT3-probe, the analyzer uses the entry-gate to transfer the control to the due destination. Stack operations are also made properly if it is a call or ret instruction. For a non-transfer instruction affected by the INT3-probe, it copies the affected instructions to the entry-gate's NOP-slide and rewrites them according to their new VAs. If the memory operand is referenced using RIP-relative addressing in the original instruction, the analyzer rewrites it with register-indirect addressing mode followed by another instruction to restore the register used in addressing. The total length of the two instructions is up to 22 bytes. Note that the transformation is necessary since the distance between the entry-gate and the probe is probably larger than 2GB which is the maximum value represented by the 32-bit signed offset.

In short, EFI probes incur much less overhead than probes in other systems since the analyzer does not need to emulate the target execution or single-step it.

5.4.3 EFI Using Probes

All probes can be installed and uninstalled at anytime by the analyzer. By choosing the timing strategy, the analyzer can make different types of EFI, which empowers it to tightly and flexibly control the target execution in user and kernel modes.

EFI With Tracing

To trace the target, the JMP-probe[3] is installed and then removed (after being executed) along a *sequence* of virtual addresses $\langle \text{VA}_1, \text{VA}_2, \cdots, \text{VA}_n \rangle$ in the target instruction flow. A requirement for successful tracing is no transfer instruction between VA_i and VA_{i+1} for $1 \leq i < n$. Hence, the tracing granularity ranges from the instruction level to the basic block level.

All the probe sites for tracing (i.e., VA_i-s) can be decided at runtime. When the target flow reaches VA_i, the control is switched to the analyzer code through

[3]In rare cases when there is no suitable segment selector for the JMP-probe, the analyzer uses the INT3-probe for tracing.

the JMP-probe and the exit-gate. After executing its analysis function, the analyzer restores the bytes replaced by the JMP-probe at VA_i, determines VA_{i+1} and installs the probe there. It then updates the entry-gate so that the target resumes execution from VA_i with original instructions. For cross-block tracing, the JMP-probe is installed at the transfer instruction of a block. When the analyzer gains the control, it evaluates the transfer destination of the instruction. Note that it inherits the target CPU context and also has native accesses to the virtual memory. Since the analyzer can pass the control to the next block via the entry-gate.

EFI With Breakpoints

The INT3-probes are installed at a *set* of virtual addresses in the target code. Whenever the target control flow reaches a probe, the analyzer gains control via the INT#3 handler and the exit-gate. The analyzer uses the aforementioned recovery technique to make up for execution of the affected instruction.

Specifically, the analyzer configures the relocated GDT, IDT and TSS to provide a new exception stack and a new handler in OASIS-Lib. When INT#3 is asserted, the hardware saves the context to the new exception stack, instead of any stack page from the target. The new handler determines whether the event is due to a probe. If true, it jumps to the exit-gate. Otherwise, it prepares the target's kernel or except stack according to the guest system setting, and then jumps to the target's own INT#3 handler.

Interrupt & Exception EFI

The target instruction stream may encounter interrupts or exceptions such as page faults. The analyzer can intercept these events and examine how they are handled. We take the page fault exception (#PF) as an example as it is often used by the kernel to manage virtual address spaces. To analyze the target kernel's page fault handling, the analyzer installs a new #PF handler to the relocated target IDT. When a page fault occurs, the hardware passes the control to the new handler which deploys the INT3-probe and/or the JMP-probe on the target handler before its execution.

5.5 Security and Transparency Assessment

The analyzer *security* means that the adversary cannot tamper with its code, data and control flow, and *transparency* means that the adversary cannot detect any analysis related artifact. No artifacts of OASIS or the analyzer is exposed to the guest and no modification is made on the guest software or hardware settings. All memory accesses from the guest are restricted to the mapped physical pages. The adversary may manipulate the target page tables which are enclosed in the analyzer-target hierarchy for accessing the target. Nonetheless, this is equivalent to feeding poisonous data to the analyzer, an indispensable risk for all dynamic analysis systems. Hence, no direct attack from the guest compromises OASIS security and transparency. Note that side-channel attacks are possible, especially those leveraging L3 cache lines.

Next, we assess security and transparency against attacks from the target thread running in the onsite environment. The onsite core's `IDTR`, `GDTR` and control registers are set inaccessible to the target in order to prevent the target from changing the system setting secretly and detecting OASIS existence. A read access to them is trapped and returned with the original value. The main attack surface is OASIS-Lib mapped in the target-lib hierarchy. The library consists of one code page (execution-only), four read-only data pages and one writable data page in the virtual address space. In addition, four paging structure pages are used in the target-lib hierarchy though they are not mapped to virtual addresses. Note that access permissions on OASIS-Lib pages are all set at the T-EPT entries, which are inaccessible to the target.

5.5.1 Security Against the Exported Target

We show that, even if the target knows the VAs used by OASIS-Lib, the target cannot compromise the analyzer security except feeding malicious data. We consider attacks that modifies OASIS-Lib data or executing OASIS-Lib code. The only writable data in OASIS-Lib is used for the exit-gate to save `RAX` and `RCX`. Hence, their illicit modifications only imply faked data in the two registers.

The target cannot modify OASIS-Lib's code page as it is not writable under T-EPT. Nonetheless, the target can jump to any instruction in it. Since the analyzer runs under A-EPT and the target runs under T-EPT, a prerequisite for a successful security attack is to switch the underlying EPT to A-EPT. The target cannot achieve its goal by using the `vmfunc` instruction in the gates, because the destination of the transfer instruction in both gates are stored in a read-only page and cannot be manipulated. The only attack surfaces left on the gates are the 22 nops on the entry gate which are used for the makeup execution. If the target successfully switches to A-EPT, those instructions may read/write into the analyzer's space.

We propose to use three instructions from line 4 to line 6 to check the index in `RCX`. There are only two valid EPTPs in the EPTP-list with index 0 and 9 pointing to T-EPT and A-EPT respectively, `vmfunc` with other indexes causes a VM exit which is captured by the OASISManager. The idea of the checking is to make line 6's transfer destination has a dependence on the value in `RCX`. There are 6 bytes in total for line 5 & 6, 36 bytes in total from line 7 to line 30. With the two `LEA` instructions in line 5 and 6, `RAX` points to line 7 if `RCX` is 0, or to line 31 if `RCX` is 9. If the target switches to A-EPT with index 9, the execution flows from line 6 to line 31 directly and finally results in an EPT violation since the destination of line 31 is the target's code which is not executable in A-EPT.

The target cannot use its own `vmfunc` instruction either, because the instruction next to `vmfunc` is not executable in A-EPT. The two boundary pages in the analyzer's 512-GB band are not mapped, thus the execution cannot sequentially flow from the target to the analyzer.

Hence, on the premise that OASIS-Lib's transparency is broken, the target can only feed the analyzer with faked `RAX` and `RCX`. Other parts of the CPU context are still genuinely fetched from the target. The analyzer security still withholds.

5.5.2 Transparency Against the Exported Target

When the target runs, all OASIS related artifacts (e.g., the GDT, IDT, and `CR3`) are protected against the target accesses by configuring the hardware, except that OASIS-Lib is protected by hiding it with randomization. To detect the existence of OASIS-Lib, the target has to guess its VAs by making read or execution references to them. Detection success is dependent on a success guess of the VA or GPA. We highlight that a wrong guess leads to an EPT violation and the EFI analysis is then aborted. Thus, the target can only make one guess.

Under the target-lib hierarchy, OASIS-Lib occupies 6 continuous pages with a random base address in the 512-GB band. Supposing that the band (i.e., λ in Figure 5.5) is known, the average success probability for one target instruction to access OASIS-Lib data or code is $6 \times 2^{12}/2^{39} \approx 2^{-24}$. More discussions are in Chapter 7.

To guess a GPA used by OASIS-Lib, the target fills in one page table entry with the guess. A successful read at corresponding VA implies a correct guess. Note that GPAs of OASIS-Lib are randomly chosen from $(T, 2^{48})$. Besides six pages for OASIS-Lib code and data, there are four paging structure pages. Hence, supposing that the guest is configured with 8 GB main memory, the success probability is merely $\frac{6+4}{(2^{48}-2^{33})/2^{12}} \approx 2^{-31}$. In short, the probability to compromise transparency is significantly low.

5.6 Case Studies

We develop three analyzers as case studies to showcase OASIS based EFI. All of them use Dyninst [85] APIs to disassemble the target binary code before analysis or on-the-fly to extract instruction level semantics. Dyninst is slightly customized to load memory-resident binaries.

5.6.1 Case I: Full-space Tracing

Since tracing is a basic function in dynamic analysis, our first case is a full-space EFI tracer using the JMP-probe, which also demonstrates OASIS capability of coping with complex kernel execution scenarios. The tracer is tested against three Linux shell commands (`ls`, `pwd`, and `kill`) and SuperPI [86] which computes 16K digits of π. While SuperPI mainly runs in user-space with a huge number of small-sized basic blocks, the Linux commands have more kernel mode execution. For each target, the tracing starts at the first user-space instruction, i.e., the entry of the default loader, and stops at the issuance of `exit` so that the target process is released and completes the exit procedure in the guest. The tracer successfully traces not only the synchronous execution of these targets, but also asynchronous executions due to events like page faults and I/O interrupt handling. For system calls, the tracer places the probe to the first block of the corresponding system call handler so as to tracing the system call handler execution. Asynchronous events are captured due to the relocated IDT. The installed handler notifies the tracer to place the probe to the target's own interrupt handler. The experiments results are reported in Table 5.1 below.

The OASIS based EFI tracer has its pros and cons as compared hardware-aided full-space tracers such as MALT [1] using PMU and Ninja [2] using ARM ETM. The main advantages is its tracing flexibility. Since the probe can be installed anywhere in the target, it can trace an arbitrary slice of instructions within a basic block. In contrast, a hardware-aided tracing is restricted to the types of events and instruc-

Target Program	# of syscalls	# of PFs	# of code pages	# of transfers	# of cross page transfer
SuperPI	68	138	283	1,674,155	302,609
ls	38	60	275	114,430	21,437
pwd	33	57	237	95,498	17,341
kill	33	55	253	94,078	17,147

Table 5.1: Tracing Report.

tions supported by the facility. Hence, it cannot make flexible intra-block tracing except single-stepping. Moreover, the EFI tracer is better at introspection and control due to its native accesses and CPU mode sharing with the target. Since hardware facility typically reports the virtual address of the monitored event only, a hardware-assisted tracers in a higher privileged environment still needs to bridge the gap to retrieve data from the target. The EFI tracer also has better performance than MALT since no hardware event is incurred by the tracer. We report the target slowdown in Table 5.2, where "preprocessing" refers to disassembling the target before tracing. Although ARM ETM does not incur overhead for tracing, it only outputs the VAs to Ninja which cannot pause and then control the target as in MALT and OASIS. Hence, Ninja's introspection to the target essentially races with the target.

	SuperPI	ls	pwd	kill
MALT [1]	192	595	134	n/a
Ninja [2]	1	n/a	n/a	n/a
OASIS w/o preprocessing	195	99	82	72
OASIS w/ preprocessing	183	77	62	54

Table 5.2: Times of slowdown on test cases

The main drawback of the EFI tracer is that all transfer instructions have to be checked to avoid losing the control, whereas the hardware facility functions as long as the configuration is not changed.

5.6.2 Case II: Kernel Analysis With Fuzzing and EFI

Kernel analysis benefits from fuzzing techniques to generate and mutate userland inputs in order to explore different kernel execution paths. As important as input fuzzing is runtime intelligence collection which is instrumental to deep understanding of the tested execution. Existing kernel fuzzing tools [87, 88, 89, 19] rely on static kernel code instrumentation to achieve it. For instance, Google Syzkaller [19] relies on the kernel address sanitizer (KASAN) [90] to report memory related operations and `ftrace` to log kernel function calls.

Code instrumentation shows two limitations under the kernel fuzzing context. Firstly, it is not adaptive enough to meet different analysis demands because the concerned code region and behaviors vary from case to case. Secondly, it is inapplicable to those dynamically loaded kernel modules that cannot be (easily) instrumented, e.g., a proprietary driver built for a production OS and a malicious module armored with anti-instrument techniques. In this case study, we use two examples to show how EFI complements code instrumentation in Syzkaller kernel fuzzing. The common approach is to export the Syz-executor process to the onsite environment so that the EFI analyzer makes on-demand analysis and acquires runtime data inaccessible to the instrumentation code. Figure 5.9 below depicts how the EFI analyzer works in tandem with Syzkaller.

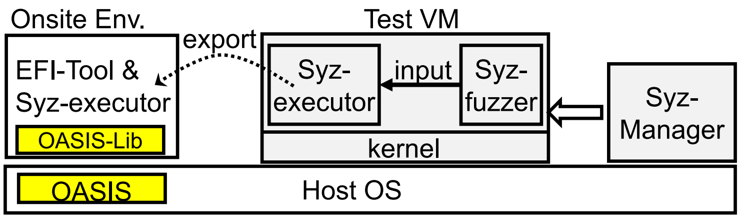


Figure 5.9: An EFI analyzer works in tandem with Syzkaller (the gray boxes).

Dynamic Postmortem Analysis

In our fuzzing test, one reported reproducible crash is caused by a page fault when KASAN accessing 0xffffef010d3415ff. The report also shows that the faulting access is to validate a memory access to `ata_bmdma_prd[pi-1]` within the kernel function `ata_bmdma_fill_sg()`.[4] To understand the build-up of the page fault,

[4]Both the function and its caller are in the default ACSI device driver in the kernel.

we develop an EFI analyzer to collect runtime data from the reproduced execution by following the strategy derived from a static analysis.

To choose the EFI instrumentation junctures, we make a backward slicing upon the instruction reading `ata_bmdma_prd[pi-1]`. We then determine the memory objects to introspect at each juncture by checking the source code correlating to the sliced instructions. The objects to collect mainly include the input parameter to `ata_bmdma_fill_sg()`, the variable `pi`, and objects the function's control flow depends on. After the Syz-executor is exported to the onsite environment, the analyzer uses the INT3-probe to monitor every `ata_bmdma_fill_sg()` invocation. When the invocation matches the one in the report, the analyzer removes the INT3-probe and traces the control flow with the JMP-probe until the page fault occurs. For blocks containing the sliced instructions, the analyzer runs a sub-block tracing on those slices. At each EFI juncture, it references and fetches the needed objects with their VAs, which means that all pointers in the kernel objects can be dereferenced directly.

Upon experiment completion, the analyzer reports the trace comprising 152 basic blocks including blocks from KCOV and KASAN. The control flow shows that the code incrementing `pi` within `ata_bmdma_fill_sg()` is never executed. Since `pi` is initialized with 0, the access to `ata_bmdma_prd[pi-1]` becomes an array underflow with index -1. Consequently, validation of this memory access causes KASAN to read a nonexistent metadata object, which triggers the page fault the kernel cannot handle. A further analysis of the collected data objects shows that `pi` is not incremented because none of the objects nested in the function input (from the fuzzer) is well-formed.

Exploration of Untrusted Driver

Our second analyzer runs with Syzkaller to uncover hidden behaviors of an un-instrumented kernel-space driver without relying on its source code. KCOV, KASAN and `ftrace` cannot report its behavior due to absence of instrumentation.

Moreover, it conceals its kernel function invocations against `ftrace` by adding 5-byte offset to the call destinations so that `ftrace` instrumentation in the callee's prologue is skipped. Specifically, the target is a synthesized malicious driver with stealthy behaviors based on a rootkit in Github [5]. When the third parameter of the driver's `ioctl` handler ends with 0xFF, the handler escalates the privilege and removes the current task from the task list.

In the experiment, Syzkaller generates the fuzzing inputs to the driver's `ioctl` while the EFI analyzer extracts the runtime information. For each exported Syz-executor, the analyzer installs an INT-3 probe at the entrance of the driver's `ioctl` handler. Since the driver is randomly loaded when the system boots up, the analyzer locates the handler at runtime via a series of introspections, starting from the current `task_struct` object in the PERCPU data structure to `files_struct`, `fdtable`, and so on until reaching the `unlocked_ioctl` object containing the driver's `ioctl` handler address. When the INT3-probe is triggered, the analyzer removes it and installs an INT3-probe at the return address. It then starts the control flow tracing within the driver's code. If a control transfer to the kernel is encountered, it stops tracing and installs another INT3-probe on the instruction which the handler is expected to resume, so that it can continue to trace the driver. The analysis ends when the driver `ioctl` returns to its kernel caller.

In the end, the analyzer successfully captures the fuzzed system call parameters triggering the hidden path. In the hidden path, the handler executes for 65 basic blocks and 7 of calls to kernel functions including `prepare_cred()` and `commit_creds()`. In the non-hidden path, there are only 7 basic blocks executed.

[5]https://github.com/croemheld/lkm-rootkit

5.6.3 Case III: Kernel Device Driver Exploit Analysis

Our third case shows an EFI analyzer that dynamically chooses what probe to be installed to what address. By using breakpoint EFI and tracing EFI in a concerted fashion, the analyzer examines how a user-space program issues `ioctl` system calls with malicious parameters to exploit vulnerable dynamically-loaded kernel drivers. The analyzer proceeds in three steps.

STEP I). The analyzer begins with installing an INT3-probe BP_1 at the entry of the `ioctl` system call handler.

STEP II). When BP_1 is triggered, the analyzer locates the driver's `ioctl` handler which the kernel's default handler is to call. It installs the second INT3-probe BP_2 at the device handler entry and makes a forward slicing on it with the third `ioctl` parameter (namely the address of the user-space supplied parameter) being the slicing criterion. All virtual addresses of instructions in the slice are saved.

STEP III). When BP_2 is triggered, the analyzer traces the basic-block control flow of the device handler execution using the JMP-probe, until encountering an `iret` which returns to userland. For those basic blocks containing instructions in the forward slice, it makes an intra-block tracing along sliced instructions to capture the data flow. For those basic blocks not in kernel space, it makes single-step tracing. All traced virtual addresses and instructions are recorded. For instructions in the slice, the analyzer also records its memory operations (if any).

We test the analyzer against a Linux malware prototype[6] which launches an ROP attack in the kernel by exploiting a vulnerable device driver. The tool's forward slicing finds 10 instructions in 2 blocks of the driver's handler and the last instruction is an indirect call. It means that the parameter determines the indirect call destination. During tracing, the analyzer finds that the destination turns out to be in user space. Its subsequent single-step tracing shows that there are 94 basic blocks executed including 8 blocks which are ROP gadgets ending with `ret` or indirect `call`. These

[6]https://github.com/vnik5287/kernel_rop

gadgets prepare parameters and invoke two kernel functions `prepare_creds()` and `commit_creds()` to elevate the privilege of current thread to the root. It also reveals that the kernel stack is replaced with a user-space stack.

Summary. The case studies demonstrate EFI's agility in controlling and introspecting a kernel thread. EFI analyzers are well-suited for targeted and fine-grained analysis as the analyst can flexibly choose when and where to deploy the INT3-probe and the JMP-probe. The case studies also show the benefits of native target accesses. Both in the postmortem analysis and case study III, the analyzer accesses the target thread's kernel object `thread_info`, `task_struct`, `files_struct`, `fdtable` etc, by directly referencing their VAs. In addition, it shows that existing analysis APIs such as instruction slicing can be conveniently applied by the EFI analyzer despite its non-conventional runtime environment. It is not difficult to develop them as they are user-space programs and can benefit from existing libraries.

5.7 Case Study Implementation Report

To support EFI analyzer implementation, we develop OASIS APIs listed in Table 5.3. Most of the functions are to instruct OASIS to facilitate the EFI session. `find_n_entry` and `find_n_exit` are entirely the analyzer's logic as they resolve target instructions by calling Dyninst APIs. Our tools uses these two functions to locate the probe site, especially for tracing. All three analyzers are quite short and tidy. Table 5.4 reports their source code sizes.

In the following, we report more implementation details of the postmortem analyzer. Table 5.5 lists the virtual address space layout of the postmortem analyzer. OASIS API functions and the analyzer are compiled and linked as one position-independent-executable binary. The analyzer code, static data, and its heap occupy around 500 KB. The shared libraries dominate the address space consumption, mainly due to Dyninst libraries.

The kernel function under analysis is `ata_bmdma_fill_sg()` which is de-

OASIS API name	Description
onsite_register_int3_handler	register a breakpoint handler on OASIS
onsite_register_trace_handler	register a tracing handler on OASIS
onsite_register_pf_handler	register a #PF handler on OASIS
onsite_wait_for_request	inform OASIS that the analyzer is ready to analyze
onsite_t_run	start/resume the target execution
onsite_end_analysis	restore the target back to guest
onsite_install_int3_probe	install an INT3-probe at the given address
onsite_install_trace_probe	install a JMP-probe at the given address
onsite_rm_int3_probe	remove the INT3-probe at the given address
onsite_rm_trace_probe	remove the JMP-probe at the given address
find_n_exit	With the given address, it returns the address of next control transfer instruction, i.e., the exit point of the current basic block.
find_n_entry	With the given address, it returns the destination if the address points to a control transfer instruction, or the address itself if it points to a non-control transfer instruction.

Table 5.3: OASIS APIs and their descriptions.

Analyzer	# of Lines of C Code
postmortem analyzer	228
untrusted driver analyzer	160
full-space control flow tracer	124

Table 5.4: source code size of analyzers.

VMA Region	Start Address	Size
Analyzer code & data	0x7ff000000000	68K
Analyzer heap	0x7ff000211000	496K
Analyzer stack	0x7ff07ffdf000	132K
System shared libraries	0x7ff010000000	40,820K
System loader ld	0x7ff020000000	144K

Table 5.5: Virtual Memory Layout of the postmortem analyzer.

fined at line 2615 of kernel source file libata-sff.c [7]. The postmortem analyzer mainly comprises three functions: a `main()` function, a breakpoint handler `ana_int3_handler()` and a tracing handler `ana_trace_handler()`. We provide below an abridged version of these functions' source code. All log related code is not shown. The objective is to shed light on the structure of the analyzer and how it controls and introspect the target thread.

Main. In the main function, the analyzer registers its handlers to OASIS and waits for OASIS to export the target to the onsite environment. When the target context is ready, it installs the INT3-probe at 0xffffffff9b5a2420, which is the address of the target function and kicks of the EFI session by yielding the CPU to the target. It only re-gains the control when the INT3-probe is triggered.

```
int main (void)
{
  // register INT3 and trace handlers
  onsite_register_handlers ();
  ....

  // inform OASIS that it is ready to analyze;
  int ret = onsite_wait_for_request ();
  if (ret){
    // install  INT3-probe
    onsite_install_int3_probe (0xffffffff9b5a2420);
    onsite_t_run (); // start to run the target;
  }
  return 0;
}
```

Breakpoint handler. When the INT3-probe is fired, the exit-gate passes the con-

[7]https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/drivers/ata/libata-sff.c?h=v4.13-rc1

trol to this handler. The analyzer first remove the probe as it is not used any more. Since the target function body is not executed yet, the handler first retrieves the addresses of local variables and then introspects the input objects by directly dereferencing the `qc` pointer which is the function input parameter. All objects are dumped into a local file in the host OS by calling `fprintf` (Line 22). As shown in the `for` loop, the analyzer traverses a kernel object list and dumps their members in the same way as in the target kernel. The analyzer then installs a JMP-probe at the VA in `probAddr` which is determined at runtime according to the basic block and the instruction slices chosen by an offline preprocessing.

```
void ana_int3_handler (void)
{
  ...
  // if it is the target function invoked
  if (target_ctx->rip == BP1) {

    //remove the INT3-probe
    onsite_rm_int3_probe(BP1);

    // target introspection
    qc = target_ctx->rdi;
    ap = qc->ap;
    prd = ap->bmdma_prd;
    ...

    //traverse and dump kernel object list
    sg = qc->sg;
    for (i = 0; i < qc->n_elem; i ++)
    {
      sg_len = sg->length;
      sg_dma_addr = sg->dma_address;
      fprintf (fp, ``sg at: %p,,... '', ...);
      sg ++;
    }

    /* find next block exit */
    blkExit = onsite_find_n_exit(BP1);

    //assign probAddr: blkExit or slice
    ...

    // install JMP-probe
    onsite_install_trace_probe(probAddr);

  }
  onsite_t_run (); //resume target;
  return;
}
```

Tracing handler. When the tracing handler is triggered by the JMP-probe, it first checks the current probe site to determine the analysis actions. As shown in the `switch`, it makes different introspections if the probe site is in the instruction slice. It then determines the next probe site by installing a new probe and resumes the target.

```
void ana_trace_handler (void)
{
    // remove JMP-probe
    onsite_rm_trace_probe(probAddr);
```

```
.....

switch (slice_idx)
{
  case 0 : //after line 2622
    addr_len = target_ctx->rbp-0x2c;
    addr_sg = target_ctx->rbp-0x38;
    addr_qc = target_ctx->rbp-0x40;
    prd = target_ctx->r15;
    pi = target_ctx->r14;
    break;
  case 1 : //after line 2632
    // acquire fresh sg pointer from stack
    sg = *addr_sg;
    sg_len = sg->length;
    sg_dma_addr = sg->dma_address;
    ....
    break;
  case 2 : //after line 2638
    len = *addr_len;
    ...
    break;
  case 3 : //after line 2641
    // read pi from R14 register
    pi = target_ctx->r14;
    prd_addr = prd[pi].addr;
    prd_flags_len = prd[pi].flags_len;
    ...
```

```
      break;
    ...
    default :
      break;
  }

  // find next block entry and exit
  blkEntry = onsite_find_n_entry (probAddr);
  blkExit = onsite_find_n_exit (blkEntry);

  //assign probAddr: blkExit or slice
  ...

  // install JMP-probe
  onsite_install_trace_probe(probAddr);
  ...
  onsite_t_run (); // resume target;
  return;
}
```

5.8 Summary

In this chapter, we introduced the notion of EFI and extended OASIS to support it. OASIS based EFI combines the advantages of hardware-trapping and code instrumentation without their disadvantages. Running in the onsite environment, an EFI analyzer uses software probes to choose the junctures for execution flow interleaving. The probes are transparent to the target. Our case studies demonstrate EFI's performance, its flexibility to control the target execution at various granularity levels and its convenience to introspect memory objects.

Chapter 6

Implementation and Evaluation

6.1 Implementation of OASIS

We have implemented OASIS on a PC with an Intel Core i5-4590 3.3 GHz processor (supporting VT-x) and 16 GB DRAM. It runs Linux kernel 3.13.0 with KVM for 64 bit x86-64 SMP. The target VM runs the same Linux version with 4 GB memory. The software of OASIS consists of 3608 SLOC for the OASIS manager, 312 SLOC for the trampoline, 77 SLOC for the `execve` handler and 90 SLOC for OASIS-Lib.

6.1.1 Launching of Analyzer

OASIS uses a *wrapper* program, a customized dynamic loader (named as the OASIS loader), and a customized `execve` handler in the host OS to launch and load the analyzer which is priorly compiled into a Position Independent Executable (PIE). The first two components are in user space while the last is in the kernel.

The wrapper program is used to launch the analyzer. When it starts, it forks out a child process which issues an `execve` system call with parameters indicating the request of running the analyzer program in onsite mode. The customized `execve` handler maps the analyzer binary together with the OASIS loader to the 512 GB virtual memory space between 0x7F8000000000 and 0x7FFFFFFFFF. In other words, we choose $\lambda_h = 011111111$. Once the starting addresses of the stack and the heap are chosen by the `execve` handler, the kernel only uses the address ranges in between for future memory allocation. Hence, all new virtual memory regions allocated at runtime fall in the same 512 GB range.

By using a debug register, the `execve` handler sets a hardware breakpoint at the entry instruction of the OASIS loader which is presumed by the kernel as the first user-space instruction to execute. When the `execve` attempts to switch to the analyzer's user mode, the debug exception is triggered and the control flow is intercepted by the trampoline. Note that since the debug exception has the higher priority than interrupts, no context switch occurs when the trampoline takes control. This ensures that the trampoline runs under the same process context as the `execve`, namely, the analyzer's.

After the analyzer thread entering to onsite mode, the mapped OASIS loader starts its execution. It maps and loads the dependent shared libraries into the designated 512 GB space before passing the control to the entry of the analyzer. Note that all system calls issued from the OASIS loader are actually handled by the host OS. Hence, there are frequent mode switches in the loading procedure.

6.1.2 OASIS Manager

The OASIS manager is implemented as a kernel module which allocates and configures needed hardware and software infrastructure for onsite mode, including the VMCS structure, EPTP-list, two sets of EPTs, O-PML4, O-PDPT, O-PD, O-PT, onsite CR3 and the mappings to the OASIS-Lib pages. In our implementation, we used the entry with prefix 111111101 in the kernel side to graft the analyzer paging structures. Namely, we set $\lambda = 111111101$. We describe the settings of control registers and descriptor tables in onsite mode first.

Control Registers. The read and write accesses to the CR3 register are configured as directly trapping to the hypervisor, this prevents the target from knowing the onsite CR3 value or switching to other threads. For the CR4 register, the FS-

GSBASE bit is set while the SMEP and SMAP bits are cleared. The read shadows for the CR4 register in onsite core's VMCS is configured with its value in the guest, thus a read access to the CR4 register returns its original value in the guest instead of the genuine value in the onsite mode's CR4. Its guest/host masks are also set accordingly to prevent the target from modifying those bits. The reason for onsite mode to set the FSGSBASE bit is to allow the analyzer conveniently switch the FS segment base address during the transitions with the target. This is crucial since the analyzer as a user space program has its own FS segment, which is distinct from the target's. The FS.base is mapped in an MSR register, while the MSR registers are only accessible in the kernel mode. The FSGSBASE bit in the CR4 register enables the instructions RDFSBASE, RDGSBASE, WRFSBASE and WRGSBASE, which allows a user mode software to read and write the FS.base and GS.base. Hence, the analyzer can switch the FS.base conveniently without escalating to the kernel privilege.

Descriptor Tables. There are two sets of IDT, GDT and TSS tables, one is for the analyzer-target and the other is for target-lib hierarchy. Those two sets of descriptor tables share the same VA while they are redirected to different physical pages in A-EPT and E-EPT. Hence, there is no need to switch these descriptor tables during transitions between the analyzer and the target. In the target-lib hierarchy, the analyzer registers its own handlers in the IDT table to capture the target's exceptions and interrupts. The TSS table is configured accordingly so that those events use the stack page prepared in the OASIS-Lib instead of the original kernel or exception stack. In the analyzer-target hierarchy, OASIS also installs stub handlers in the IDT table, so that all exceptions are trapped to the stub handler which saves the context and issues a hypercall to notify the trampoline. For the #PF stub handler, it delivers the #PF event to the analyzer if the faulting address in the CR2 register belongs to the target's address range. This happens when the analyzer's access to the target page violates with the access permissions on the GPT, e.g., placing a probe on a non-present page. Those events are handled by the analyzer instead of the host OS.

In EFI, the target thread is captured and exported to onsite mode for analysis. The CPU context is saved and restored during the transitions of the target and the analysis. We describe some of the implementation details below.

Thread Capturing. We implemented multiple ways to capture the target thread. In the LMbench benchmark testing experiments, onsite core continuously captures guest threads by sending IPIs to the guest core. Due to the IPI, the guest traps to the host OS and its current thread is exported to onsite mode for analysis. In the Syzkaller EFI case studies, we modify the syz-executor so that it issues a hypercall before it tests the `ioctl` system call. For those anti-analysis experiments, we hook the guest kernel's `execve` handler so that it issues a hypercall if it is about to launch the target program. The hypercall notifies the OASIS Manager to export the thread to onsite mode.

Thread Exportation & Restoration. For every thread exported to onsite mode, OASIS Manager clones its PML4 page to O-PML4 except the λth entry. The kernel stacks configured in the TSS table are also replicated to the onsite mode's TSS except the one reserved by the OASIS-Lib. After thread exportation, the target's PML4 page is marked as Read-only on the target VM's EPT. Any updates are synchronized to O-PML4 if there is no confliction with the chosen λth entry. The updates happen during the target thread's initialization. Upon the analyzer's request to restore the target, OASIS updates the guest core's VMCS structures using onsite core's context, including general purpose registers, XMM resgisters and MSR registers. The target continues its execution in the guest from the new context.

Context Saving & Restoring. The general purpose registers, RFLAGS, XMM registers, FS and GS base address are saved and restored during the transitions between the analyzer and the target. Since MSR registers, e.g., IA32_KERNEL_GS_BASE MSR, are only accessible in the kernel mode, they are saved and restored only if the target runs in the kernel mode. Note that the analyzer inherits the privilege from the target. The only exception is the IA32_LSTAR MSR which stores the destination address of SYSCALL instruction. The analyzer and the target share the same value for the IA32_LSTAR MSR, which is the guest kernel's system call entrance. Since that location is not executable in the A-EPT, the system calls issued by the analyzer trap to the trampoline due to an EPT violation.

EPT Synchronization. The target VM's EPTs can be updated at runtime by Linux KVM. To ensure the consistency at the EPT level, the OASIS manager hooks the KVM's EPT management routines so that any EPT update on the target is also replicated for onsite mode. The EPT management routines are `mmu_set_spte` and `remap_remove`, they are responsible for creating new entries or clearing existing entries on the EPT respectively.

6.1.3 Trampoline

The trampoline is also implemented as a kernel module, which is hooked to the host OS's debug exception pointer and the VM Exit handler. Proper filters are added so that the trampoline only handles OASIS related debug exception and VM Exit. The bulk of its workload is on handling VM Exit.

VM Exit. The trampoline uses data stored in the VMCS structure of the onsite mode environment to identify the event type (system call, page fault and general exception/interrupt). It prepares the context for the host OS accordingly, including the kernel stack for the analyzer process as well as all relevant general and control registers in the physical CPU.

For system calls, it copies `RAX`, `RDI`, `RSI`, `RDX`, `R10`, `R8`, `R9` registers from the VMCS to the physical registers, where `RAX` contains the system call number and others store parameters (if any). For exceptions, it pushes `RIP`, `RFLAGS`, `RSP`, `CS` and `SS` from the context saved by onsite mode's stub handler to the host's kernel stack in case of a general exception. In case of a #PF, it also copies the saved `CR2` to the physical `CR2` which is supposed to store the faulting linear address.

The trampoline uses the processor's debug registers to set a breakpoint, which is execution-triggered and local to the analyzer task. The breakpoint is set at the virtual address of the instruction which the analyzer resumes execution. For exceptions, it uses the address in `RIP` which points to the faulting instruction; for system calls, it uses the address in `RCX` which points to the instruction next to the system call instruction. Since Linux does not use the processor's task switch facility, the local breakpoint is in fact valid for all processes in the host. Nonetheless, the likelihood of being triggered by other processes remains slim, because the 512 GB linear address at the base 0x7F8000000000 is rarely used by others.

In the end, the trampoline uses a `jump` instruction to pass the control to the kernel exception/interrupt/system call handler. The kernel handler locates its stack based on the Per-processor Data Area (PDA) and processes the event as if it originates in the analyzer's userland.

Debug Exception. Once the debug exception is triggered due to the priorly set breakpoint, the trampoline's task is to start or resume the analyzer thread in onside mode. With the hardware's Interrupt Stack Table (IST), the trampoline handles the debug exception with a different stack from the analyzer's kernel stack. (The interrupt stack is also the one used during handling VM exit.) It ensures that the kernel stack used for handling the analyzer events is not contaminated by the trampoline execution.

Note that the breakpoint is triggered after the `iret` or `sysret` in the OS's handler is executed successfully. Hence, the kernel objects relevant to the analyzer thread reflect that the thread has entered into user mode. The execution of the trampoline after the debug exception, which is still under the analyzer thread's context, is transparent to the rest of kernel. Hence, entering or exiting onsite mode preserves the analyzer thread context in the OS's view.

6.2 OASIS Performance

To better understand OASIS's performance, we measure the CPU time taken by relevant hardware events and instructions in our platform and report them in Table 6.1. Note that the time does not cover software processing.

INT3	vmfunc	iret	VM Exit & Enter
340	147	367	916

Table 6.1: CPU cycles for relevant events and instructions

We first study the performance of OASIS (its support to analyzer application, robustness and performance), then we study the CPU time spent by OASIS for EFI.

6.2.1 OASIS Support to Analyzer

We first verify whether the host OS views the analyzer as a regular thread though it runs in onsite mode. In the experiment, the launcher program issues `getpid` in the host while the analyzer issues `getpid` in onsite mode. The result shows that the launcher program's Process ID (PID) is 4617 while the analyzer gets 4618. It corroborates the correctness of OASIS as the analyzer in onsite mode is indeed the child process of the launcher.

Although the analyzer in onsite mode receives the same service in host mode, its system calls take longer time than in a native environment due to the overhead of OASIS mode switches. To understand the actual performance drop, we measure the execution of five commonly used system calls in onsite mode and normal user mode. As shown in Table 6.2, the average overhead of system calls from onsite mode is about 2.21 μs.

System Call	`open`	`read` (4KB)	`write` (4KB)	`brk`	`getpid`
User mode	1.41	0.79	1.14	0.84	0.39
Onsite mode	3.55	2.97	3.50	3.04	2.56
Overhead	2.14	2.18	2.36	2.20	2.17

Table 6.2: Popular system call time in two modes (in μs)

We also experiment OASIS with three off-the-shelf applications: SVM[1], a program using OpenSSL library to encrypt and decrypt files, and a network program to evaluate the overhead of program loading and the correctness of system call and signal support. It takes about 20.31 milliseconds, 32.27 milliseconds and 16.39 milliseconds to launch SVM, OpenSSL, and SSH respectively. The dominant cost contains two parts. One is to initialize the needed VMCS structure and EPTs; and the other is to handle the system services requested by the loader to find and map shared libraries. These experiments show that OASIS correctly supports file operations (used by SVM and OpenSSL), keyboard I/O (used by OpenSSL and SSH) and network I/O (used by SSH).

6.2.2 OASIS EFI Transition Performance

Now, we study the EFI overhead characterized by the average round trip transition time, i.e., the time taken by the control flow to depart from the target to the analyzer and return back. Following the method in SPIDER [7], we measure the overall time difference between the target execution without being traced and execution with a null EFI tracer which has no payload function, and then divide it by the number of round-trips. The result is in Table 6.3, together with the data reported in the literature for the hypervisor-based approach [7] and the SMM-based approach [1]. The dominant overhead of using the JMP-probe includes the `vmfunc` instruction, restoring the target instruction and probe relocation (391 cycles), as well as the analyzer's CPU context saving and restoration (90 cycles). Our technique is 3.8 times faster than those in SPIDER [7]. The overhead SPIDER is attributed to the hassle of single-stepping the restored instruction while the overhead in MALT is entirely due to the hardware.

We also measure the overhead of using the INT3-probe for EFI breakpoints. It comprises of the INT#3 exception, the make-up execution of the affected instruction

[1]LIBSVM, https://www.csie.ntu.edu.tw

Transition	JMP-probe	INT3 Trap to Hyp [7]	Trap to SMM [1]
Cost	836	3,217	28,128

Table 6.3: Round-trip transition overhead using JMP-probe (in CPU cycles).

as well as control transferring between the target and the analyzer. The exception overhead varies with the probe privilege, i.e., in Ring 0 or Ring 3, while the make-up overhead varies with the types of the overwritten instructions since different methods are used for make-up execution. Table 6.4 below reports our experiment results. Note that the largest overhead (i.e., a probe on a user-space non-transfer RIP-relative instruction) is still less than 50% of the cost in SPIDER. The main reason is that, benefiting from the native-access feature of EFI, the make-up execution does not require single-stepping.

	Type of Overwritten Instruction		
	Transfer	Non-transfer RIP-relative	Others
From Ring 3	1100	1286	1280
From Ring 0	669	935	934

Table 6.4: Round-trip transition overheads using INT3-probe (in CPU cycles).

6.3 Benchmark Testing

We evaluate the impact of EFI upon the guest kernel, by running the LMbench tools [91] in the guest with and without OASIS based EFI. When the benchmark tools are running in the guest, a randomly chosen guest kernel thread is captured and exported to the onsite environment for analysis. We conduct the experiments with two analyzers. One is a null analyzer which releases the thread without any analysis and the other traces one basic block execution before releasing it. The first experiment measures the performance impact due to target thread exportation and restoration while the second assesses the overall effect due to prolonged target execution. Although the analyzer does not consume hardware resources of the guest, the slowdown of the captured thread may affect others due to synchronization or resource sharing.

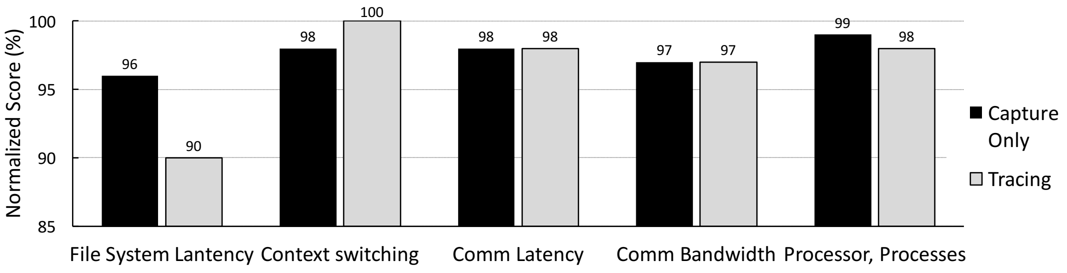


Figure 6.1: Normalized LMbench results (in % of the native benchmark results).

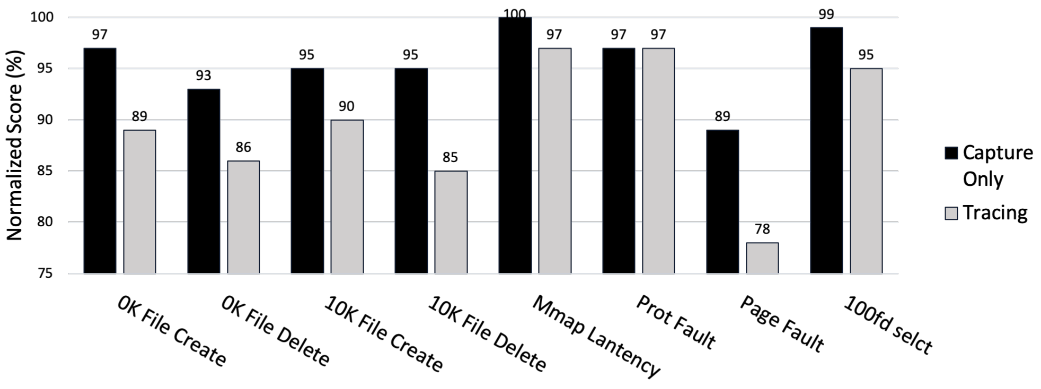


Figure 6.2: Normalized LMbench results on file systems (in % the native benchmark results).

The normalized results on all system aspects of LMbench in two experiments are shown in Figure 6.1. Except File System Latency showing significant performance slowdown (4% and 10% in two experiments), other four benchmarks only report up to 3% drop. Figure 6.2 reports the detailed data in File System Latency subcategory. The deeper investigation reveals that the performance slowdown on file system latency is largely due to page fault handling whose performance drops 11.3% and 22.3% in two experiments, respectively. One plausible reason is frequent guest EPT updates during file creation, deletion and page fault handling. Every guest EPT update has to be cloned to A-EPT and T-EPT.

6.4 Transparency Experiments

By and large, there exist three attack strategies to detect and evade dynamic analysis. The first is to discover environment related evidences such as existence of an emulator; the second is to discover software state related artifacts such as unexpected code and data modifications [27]. The third is to obstruct analysis by code and data obfuscation, usually via packing/unpacking [92]. A sophisticated malware may use a combination of them.

6.4.1 System Environment Detection

Existing techniques include anti-emulation, anti-VM, anti-sandbox and anti-debugging, targeting different dynamic analysis systems. Since OASIS does not rely on emulation or sandbox, anti-emulation and anti-sandbox attacks are irrelevant. OASIS is dependent on memory virtualization and its implementation is upon Linux KVM. Hence, anti-VM attacks can detect the KVM setting. However, we argue that with the wider adoption of virtualization in personal computers and the trend of cloud computing, the anti-VM threat is diminishing because the mere presence of a virtual machine becomes a less strong indicator of dynamic analysis than the past years.

We run experiments against anti-debugging attacks. We test OASIS against Pangu[2], an off-the-shelf program implementing popular anti-debugging techniques used by Linux malware [93, 94, 95]. We analyze it using both EFI tracing and EFI breakpoints. The results are in Table 6.5.

The first four attacks cannot detect any artifacts of the EFI analyzer. The Sigtrap attack checks whether malware receives its own `SIGTRAP` signal which is delivered to the GDB if it is under debugging. OASIS allows the target in the onsite environment to deliver and receive signals in the same way as in the guest. Hence, the attack cannot detect OASIS. We consider the `rdtsc` attack as one type of side-channel

[2]https://github.com/jvoisin/pangu

Technique	Brief description	Status
`envvar`	Checks for GDB specific environment through getenv()	PASS
`ptrace`	Checks whether the program is ptraced or not through ptrace system call	PASS
`sigtrap`	Checks whether the program receives a `SIGTRAP` signal	PASS
`parent`	Checks the program's parent name through /proc/getppid()/status and /proc/getppid()/cmdline	PASS
`rdtsc`	Checks whether the time taken of a loop exceeds a pre-specified bound	NO

Table 6.5: Anti-debugging techniques.

attacks that infer the analyzer presence by observing behavioral patterns (e.g., prolonged execution time), and leave it as future work to adjust timers and counters of the platform. In general, it remains as an open problem to systematically nullify all side-channel attacks. As noted in MALT [1], an ultimate solution may require fine-grained instruction level semantics.

6.4.2 Software State Detection

This strategy is mainly used by malware to detect code instrumentation. The most straightforward way is to compute a fingerprint of the code in the virtual address space and compare the outcome with the expected value. We test OASIS against the self-checksum program[3] which calculates a hash digest of its executable and read-only data memory. Although we install a few INT3-probes in its code, it still reports its original checksum because of our techniques in 5.4.1. We also test the self-checksum program in GDB with and without setting a BP at its main function as a comparison. They output two different values which are also different from the original checksum. This proves that the self-checksum program can detect the environment modification as well as INT-3 installed by GDB

Kirsch et. al. [27] propose thirteen methods to detect DBI tools including Pin

[3]https://github.com/Barro/self-checksum

[21], DynamoRIO [22] and Valgrind [20]. We test OASIS against their methods integrated in *jitmenot* [56]. Our EFI tracing and breakpoint tools pass all the detection methods listed in Table 6.6.

Technique	Brief description	Status
`enter`	Checks whether enter instruction is legal and can be executed	PASS
`fsbase`	Checks if `fsbase` value is the same using `rdfsbase` and `prctl`	PASS
`nx`	Tries to execute code on a non-executable page	PASS
`ripfxsave`	Executes fxsave instruction and chekcs the saved rip value	PASS
`ripsiginfo`	Causes an int3 and checks the saved rip value in fpregs	PASS
`ripsyscall`	Checks whether rip value is saved in rcx after a syscall	PASS
`smc`	Check whether the framework detects self-modifying code	PASS
`jitbr`	Detect time overhead when a conditional branch is jitted	PASS
`jitlib`	Detects JIT compiler overhead when a library is loaded	PASS
`envvar`	Checks for Pin specific environment variables on stack	PASS
`pageperm`	Checks for pages with `rwx` permissions	PASS
`vmleave`	Checks for known code patterns(VMLeave)	PASS
`mapname`	Checks mapped files's name for known values (pinbin, vgpreload)	PASS

Table 6.6: representative anti-instrument techniques.

The methods are grouped to three categories [27]. The first seven techniques are in the code cache/instrumentation artifacts category, which checks whether the code is the original code. These techniques can be applied to static binary instrumentation as well. They cannot break OASIS transparency, because OASIS does not change the target code. When probes are used, OASIS ensures that the target's view to its code remains unchanged. The next two techniques (i.e., **`jitbr`** and **`jitlib`**) detect execution time changes due to JIT compilation. EFI tracing does incur longer delay to control transfering. However, it is not detected by the tools because the EFI overhead is much shorter than using JIT. The last four techniques are in the category of runtime environment artifacts. Although OASIS relocates the target's IDT, GDT and TSS, they are invisible to the target.

6.4.3 Packer

According to Cozzi et. al. [92], the most popular packer used by Linux malware is UPX [96] which is a Type-I packer [97]. Since unpacking is essentially malware's dynamic modification in its own virtual address space, it has no adverse impact on OASIS. Note that the hierarchies used in the onsite-environment is designed to be consistent with the one in the guest and any EPT update in the guest is also cloned to the onsite-environment. However, the EFI probes are dependent on the target virtual address space. The unpacking process may read or write the page with probes or change the page's mapping to another physical page. We have explained how OASIS protects probes in these two scenarios in 5.4.1, by leveraging d-TLB and i-TLB to cope with read/write accesses and by monitoring GPA update to cope with mapping changes.

To check whether EFI tools can deal with packers, we apply UPX on a Linux shell command program `uname` which displays the system identification information. We test EFI breakpoint and tracing on the packed uname. In both experiments, the packed program runs successfully and the analyzer also achieves the intended analysis goal. In the breakpoint experiment, we place the INT3-probe at address 0x4016ab where the struct `utsname` is stored on the current stack top after being updated by the kernel. For EFI-tracing, the analyzer obtains all basic block transfers, including the unpacking procedure. The tracing results are in Table 6.7.

Target	No. of basic blocks	No. of syscalls	No. of #PF	No. of self-write bebaviors
Packed uname	311,994	38	51	388

Table 6.7: EFI-tracing for UPX packed uname.

Since our current OASIS implementation is for Linux guests, we are unable to test it against packers on Windows which have more complex packing schemes, e.g., more rounds of unpacking. Nonetheless, we foresee that Windows packers cannot compromise transparency either, because their system-level building blocks are the same as UPX, i.e., page permission changing and code modification, despite of using a more complex application-level logic.

6.5 Summary

In this chapter, we implemented OASIS and evaluated its performance as well as security with benchmarks and experiments. OASIS has robust support to a variety of pre-compiled analyzer applications with acceptable overhead. The highest transition cost of its EFI is still less than 50% of the cost in SPIDER. The performance improvement comes from twofold: 1) no CPU mode/privilege switches during the execution flow transition; 2) the consistent environment as in the guest makes the makeup execution efficient. OASIS and its EFI tools remain transparent and effective against target programs equipped with anti-analysis techniques.

Chapter 7

Discussion

Target Control Without Probes. Hardware based event-trapping methods can also be applied in OASIS to control the target execution. Hardware facilities such as PMU and debug registers can be utilized by the analyzer in a similar way to MALT [1] and Ninja [2]. Since OASIS allows the analyzer to replace the target's system level structures including IDT, interrupts triggered by PMU and debug registers can be directly handled by the analyzer. To ensure transparency, the exported target thread should be prevented from accessing PMU or debug registers. OASIS can use virtualization techniques to configure the onsite core's VMCS so that any access to those relevant registers are trapped to the hypervisor or delivered to the analyzer as a virtual exception.

The probe-based EFI in 5.4 helps the analyzer to actively tame the target execution in order to acquire fine-grained software semantics, while the trapping-based EFI is more suitable for event-centric responsive analysis. The two EFI styles only differ in target control, i.e., how the interleaving juncture is chosen and triggered. As compared with MALT and Ninja, the EFI analyzer still enjoys the advantage of native-accesses.

ARM Platform. The design of OASIS is also applicable to ARM platforms supporting virtualization extension. In ARM virtualization, virtual addresses are translated to host physical addresses through the Stage I translation table managed by the kernel and the Stage II translation table managed by the hypervisor. Similar to the guest kernel page table in an x86 platform, the Stage I translation table also uses guest physical addresses. Hence, the analyzer-target hierarchy and the target-lib hierarchy can also be constructed on an ARM platform, which is the foundation for the onsite environment.

The key difference is that the ARM architecture does not have a user-space instruction equivalent to Intel's `vmfunc` which switches the underlying Stage II translation table. Hence, interleaving the target and the analyzer instruction flows mandates privilege level switches (or exception level switches in ARM terminology). Nevertheless, due to different virtualization strategies, the switch consumes much faster (about 300 CPU cycles) than VM-exit and VM-entry in x86-86 platforms.

Security. Here we discuss one scenario where the target aggressively occupies all PML4 entries by mapping at least one page there. Then there is no available PML4 entry left for the analyzer/OASIS-Lib. OASIS can choose the λ-th entry from O-PDPT or O-PD instead of from O-PML4 to graft the paging structures for the analyzer and OASIS-Lib. If that is the case, at most three target's paging structure pages should be set as write protect in the guest VM's EPT to synchronize the modifications into O-PML4, O-PDPT, and O-PD. One entry from O-PD specifies 2MB continuous VA region, OASIS can use multiple O-PD entries for the analyzer if one is not enough. Loading the analyzer's code, stack, libraries into different VA regions does not have technique issues since that is the normal case for an application. There are at most 2^{27} O-PD entries, the attacker needs 512GB (2^{27}*4K) physical memory in the platform if it wants to occupy all of O-PD entries. It is impractical. Another attacking scenario is that the target wants to dynamically guess the λ chosen by OASIS by mapping one page within the VA range specified by λ. The essence of this attack is the same as the previous one thus share the same solution.

Chapter 8

Dissertation Conclusion and Future Directions

8.1 Conclusion

In this dissertation, we designed a system infrastructure named OASIS. The key feature of OASIS is to fuse the target's paging hierarchy with the analyzer by using virtualization techniques. OASIS offers three capabilities to the analyzer: to natively access the target's live virtual memory with mapping consistency; to dynamically control and instrument the target execution; and to receive transparent system services from the host OS.

We also proposed two new dynamic analysis models on top of OASIS . In OMA, the tools successfully collect the target program's virtual memory intelligence and its system call events via conducting large-scale and intensive target memory accesses. In EFI, the analyzer natively introspects and instruments the target's execution flow with isolation and transparency protection. The case studies in EFI demonstrate its performance and the great flexibility to control the target execution at various granularity levels. Developing tools on top of OASIS is easy since they are normal user space applications despite the capability for kernel analysis.

We implemented a prototype of OASIS and conducted experiments to evaluate its security and performance. OASIS provides robust OS supports to a full-fledged analyzer application with acceptable overhead. It remains effective and transparent towards the target programs which are equipped with anti-analysis techniques.

8.2 Future Directions

In the future, we aim to extend OASIS to support more analysis scenarios. Followings are the possible directions.

Testing and Analysis. The onsite environment has the potential to be extended and applied for kernel testing by leveraging the hybrid paging hierarchies. OASIS can be used to either test the target code or analyze the target data. To test an untrusted kernel, the tester program in the analyzer-target hierarchy supplies the testing inputs to the kernel thread. It then uses EFI to introspect the target thread to collect the testing results. A potential application of such code tests is to conduct kernel concolic execution [98] which is mainly realized using QEMU (e.g., the popular S2E platform [99]) as of now.

Although the OASIS design has the architectural support for such tests, there remain several challenges. One of the main hurdles is how to efficiently and effectively confine the impact of the tests within the onsite environment, instead of on the guest. Another is how to handle subject crashes and roll-backs.

Paralleled Analysis. The onsite environment we have described so far has a single core which is occupied by the analyzer and the target alternately. The analyzer cannot gain the control until the target "relinquishes" the core due to the probes, exceptions or external interrupts. The single-core setting does not allow the analyzer to proactively control the target. To overcome with this limitation, OASIS can be extended to launch a multicore onsite environment for two or more analyzer threads. One potential application is to run two analyzer threads with two target threads. The two analyzer threads can coordinate with each other to tune the timing of execution in order to trigger a racing condition vulnerability in the target threads. Another application is to run two analyzers upon the same target with one for monitoring and the other for operation. The monitoring thread persistently occupies one core to monitor events in the target execution. When needed, it sends an IPI to preempt the target execution so that the operation thread makes due analysis.

Bibliography

[1] F. Zhang, K. Leach, A. Stavrou, H. Wang, and K. Sun, "Using hardware features for increased debugging transparency," in *Proceedings of IEEE Symposium on Security and Privacy (S&P)*, 2015.

[2] Z. Ning and F. Zhang, "Ninja: Towards transparent tracing and debugging on ARM," in *Proceedings of USENIX Security Symposium*, 2017.

[3] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A fast address sanity checker," in *Proceedings of the 2012 USENIX Annual Technical Conference (USENIX ATC)*, 2012, pp. 309–318.

[4] P. P. Bungale and C.-K. Luk, "PinOS: a programmable framework for whole-system dynamic instrumentation," in *Proceedings of the 3rd International Conference on Virtual Execution Environments*, 2007.

[5] J. Zeng, Y. Fu, and Z. Lin, "Pemu: A pin highly compatible out-of-vm dynamic binary instrumentation framework," in *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2015, pp. 147–160.

[6] 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)*, 2008.

[7] Z. Deng, X. Zhang, and D. Xu, "SPIDER: Stealthy binary program instrumentation and debugging via hardware virtualization," in *Proceedings of the 38th Annual Computer Security Applications Conference (ACSAC)*, 2013.

[8] S. Zhao, X. Ding, W. Xu, and D. Gu, "Seeing through the same lens: Introspecting guest address space at native speed," in *Proceedings of the 26th USENIX Security Symposium*, 2017.

[9] M. H. Ligh, A. Case, J. Levy, and A. Walters, *The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux and Mac Memory*. John Wiley & Sons, 2014.

[10] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee, "Virtuoso: Narrowing the semantic gap in virtual machine introspection," in *Proceedings of IEEE Symposium on Security and Privacy (S&P)*, 2011, pp. 297–312.

[11] A. Saberi, Y. Fu, and Z. Lin, "Hybrid-bridge: Efficiently bridging the semantic gap in virtual machine introspection via decoupled execution and training memoization," in *Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA*, 2014.

[12] D. Jang, H. Lee, M. Kim, D. Kim, and B. B. Kang, “ATRA: Address translation redirection attack against hardware-based external monitors,” in *Proceedings of the 2014 ACM Conference on Computer and Communications Security (CCS)*, 2014.

[13] H. Moon, H. Lee, J. Lee, K. Kim, Y. Paek, and B. B. Kang, “Vigilare: toward snoop-based kernel integrity monitor,” in *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS)*, 2012.

[14] F. Zhang, J. Wang, K. Sun, and A. Stavrou, “Hypercheck: A hardware-assisted integrity monitor,” *IEEE Transactions on Dependable and Secure Computing (TDSC)*, vol. 11, no. 4, 2014.

[15] B. Jain, M. B. Baig, D. Zhang, D. E. Porter, and R. Sion, “SoK: Introspections on trust and the semantic gap,” in *Proceedings of the 35th IEEE Symposium on Security and Privacy*, 2014.

[16] A. Vasudevan and R. Yerraballi, “Cobra: Fine-grained malware analysis using stealth localized-executions,” in *Proceedings of the 27th IEEE Symposium on Security and Privacy (S&P)*, 2006.

[17] B. Chamith, B. J. Svensson, L. Dalessandro, and R. R. Newton, “Instruction punning: Lightweight instrumentation for x8664,” in *Proceedings of ACM Conference on Programming Language Design and Implementation (PLDI)*, 2017.

[18] P. Feiner, A. D. Brown, and A. Goel, “Comprehensive kernel instrumentation via dynamic binary translation,” in *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[19] D. Vyukov. (2015) Syzkaller. [Online]. Available: https://github.com/google/syzkaller

[20] N. Nethercote and J. Seward, “Valgrind: a framework for heavyweight dynamic binary instrumentation,” *ACM Sigplan notices*, vol. 42, no. 6, pp. 89–100, 2007.

[21] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, “Pin: building customized program analysis tools with dynamic instrumentation,” in *Acm sigplan notices*, vol. 40, no. 6. ACM, 2005, pp. 190–200.

[22] D. Bruening, T. Garnett, and S. Amarasinghe, “An infrastructure for adaptive dynamic optimization,” in *International Symposium on Code Generation and Optimization, 2003. CGO 2003.* IEEE, 2003, pp. 265–275.

[23] F. Falc and N. Riva, “Dynamic binary instrumentation frameworks: I know you’re there spying on me,” in *REcon*, 2012.

[24] X. Li and K. Li, “Defeating the transparency features of dynamic binary instrumentation,” in *BlackHat USA*, 2014.

[25] K. Sun, X. Li, and Y. Ou, “Break out of the truman show: Active detection and escape of dynamic binary instrumentation,” in *Blackhat Asia*, 2016.

[26] M. Polino, A. Continella, S. Mariani, S. D’Alessio, L. Fontana, F. Gritti, and S. Zanero, “Measuring and defeating anti-instrumentation-equipped malware,” in *Proceedings of the 14th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2017.

[27] J. Kirsch, Z. Zhechev, B. Bierbaumer, and T. Kittel, "PwIN - pwning intel pin: Why DBI is unsuitable for security applications," in *Proceedings of European Symposium on Research in Computer Security (ESORICS)*, 2018.

[28] D. C. D'Elia, E. Coppa, S. Nicchi, F. Palmaro, and L. Cavallaro, "Sok: Using dynamic binary instrumentation for security (and how you may get caught red handed)," in *Proceedings of AsiaCCS*, 2019.

[29] L. Szekeres, M. Payer, T. Wei, and D. Song, "SoK: Eternal war in memory," in *Proceedings of the 34th IEEE Symposium on Security and Privacy*, 2013.

[30] Intel Corporation, *Intel© 64 and IA-32 Architectures Software Developer's Manual*, December 2016, vol. 3.

[31] W. Li, Y. Xia, H. Chen, B. Zang, and H. Guan, "Reducing world switches in virtualized environment with flexible cross-world calls," *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3S, pp. 375–387, 2015.

[32] W. Tang and Z. Mi, "Secure and efficient in-hypervisor memory introspection using nested virtualization," in *2018 IEEE Symposium on Service-Oriented System Engineering (SOSE)*. IEEE, 2018, pp. 186–191.

[33] Z. Mi, D. Li, Z. Yang, X. Wang, and H. Chen, "Skybridge: Fast and secure interprocess communication for microkernels," in *Proceedings of the Fourteenth EuroSys Conference*, 2019, pp. 1–15.

[34] M. Hedayati, S. Gravani, E. Johnson, J. Criswell, M. Scott, K. Shen, and M. Marty, "Janus: Intra-process isolation for high-throughput data plane libraries," Technical Report UR CSD/1004, Tech. Rep., 2018.

[35] J. Nakajima, J. Tsai, M. Ergin, Y. Zhang, and W. Wang, "Extending kvm models toward high-performance nfv," *Intel Corporation, Oct*, vol. 14, p. 28, 2014.

[36] V. P. Kemerlis, G. Portokalidis, and A. D. Keromytis, "kguard: lightweight kernel protection against return-to-user attacks," in *21st USENIX Security Symposium (USENIX Security)*, 2012, pp. 459–474.

[37] Z. Hua, D. Du, Y. Xia, H. Chen, and B. Zang, "EPTI: Efficient defence against meltdown attack for unpatched vms," in *2018 USENIX Annual Technical Conference (USENIX ATC)*, 2018, pp. 255–266.

[38] C. Willems, T. Holz, and F. Freiling, "Toward automated dynamic malware analysis using cwsandbox," *IEEE Security & Privacy*, vol. 5, March 2007.

[39] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike, "Enforcing forward-edge control-flow integrity in GCC & LLVM," in *23rd USENIX Security Symposium (USENIX Security)*, 2014, pp. 941–955.

[40] G. J. Duck and R. H. Yap, "Effectivesan: type and memory error detection using dynamically typed c/c++," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2018, pp. 181–195.

[41] M. Wenzl, G. Merzdovnik, J. Ullrich, and E. Weippl, "From hack to elaborate techniquea survey on binary rewriting," *ACM Computing Surveys (CSUR)*, vol. 52, no. 3, pp. 1–37, 2019.

[42] M. Prasad and T.-c. Chiueh, "A binary rewriting defense against stack based buffer overflow attacks." in *USENIX Annual Technical Conference, General Track*, 2003, pp. 211–224.

[43] E. Bauman, Z. Lin, K. W. Hamlen *et al.*, "Superset disassembly: Statically rewriting x86 binaries without heuristics." in *NDSS*, 2018.

[44] D. W. Wall, "Systems for late code modification," in *Code GenerationConcepts, Tools, Techniques*. Springer, 1992, pp. 275–293.

[45] A. Srivastava and A. Eustace, "Atom: A system for building customized program analysis tools," in *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, 1994, pp. 196–205.

[46] D. Andriesse, X. Chen, V. Van Der Veen, A. Slowinska, and H. Bos, "An in-depth analysis of disassembly on full-scale x86/x64 binaries," in *25th USENIX Security Symposium*, 2016, pp. 583–600.

[47] C. Eagle, *The IDA pro book.* no starch press, 2011.

[48] R. Wang, Y. Shoshitaishvili, A. Bianchi, A. Machiry, J. Grosen, P. Grosen, C. Kruegel, and G. Vigna, "Ramblr: Making reassembly great again." in *NDSS*, 2017.

[49] A. Di Federico, M. Payer, and G. Agosta, "REV. NG: a unified binary analysis framework to recover cfgs and function boundaries," in *Proceedings of the 26th International Conference on Compiler Construction*, 2017, pp. 131–141.

[50] R. R. Branco, G. N. Barbosa, and P. D. Neto, "Scientific but not academical overview of malware anti-debugging, anti-disassembly and anti-vm technologies," *Black Hat*, vol. 1, 2012.

[51] B. Buck and J. K. Hollingsworth, "An api for runtime code patching," *The International Journal of High Performance Computing Applications*, vol. 14, no. 4, pp. 317–329, 2000.

[52] J. Newsome and D. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," in *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS)*, 2005.

[53] M. Hron and J. Jermář, "Safemachine: Malware needs love, too," *Virus Bulletin*, 2014.

[54] A. Bougacha. (2012) Detecting valgrind. [Online]. Available: http://repzret.org/p/detecting-valgrind/

[55] F. Falc and N. Riva. (2012) Extensible anti-instrument tester(exait). [Online]. Available: https://www.coresecurity.com/corelabs-research/open-source-tools/exait

[56] J. Kirsch, Z. Zhechev, B. Bierbaumer, and T. Kittel. [Online]. Available: https://github.com/zhechkoz/PwIN/tree/master/jitmenot

[57] A. Tamches and B. P. Miller, "Fine-grained dynamic instrumentation of commodity operating system kernels," Ph.D. dissertation, University of Wisconsin–Madison, 2001.

[58] D. J. Pearce, P. H. Kelly, T. Field, and U. Harder, "GILK: A dynamic instrumentation tool for the linux kernel," in *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*. Springer, 2002, pp. 220–226.

[59] H. Yin and D. Song, "Temu: Binary code analysis via whole-system layered annotative execution," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2010-3*, 2010.

[60] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," *ACM SIGOPS operating systems review*, vol. 37, no. 5, pp. 164–177, 2003.

[61] Y. Fu, J. Zeng, and Z. Lin, "HYPERSHELL: A practical hypervisor layer guest OS shell for automated in-vm management," in *2014 USENIX Annual Technical Conference (USENIX ATC)*, 2014, pp. 85–96.

[62] A. Srivastava and J. Giffin, "Efficient monitoring of untrusted kernel-mode execution," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2011.

[63] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "kafl: Hardware-assisted feedback fuzzing for OS kernels," in *26th USENIX Security Symposium*, 2017, pp. 167–182.

[64] H. Chen, D. Dean, and D. Wagner, "Model checking one million lines of c code," 2004.

[65] H. H. Feng, J. T. Giffin, Y. Huang, S. Jha, W. Lee, and B. P. Miller, "Formalizing sensitivity in static analysis for intrusion detection," in *Proceedings of IEEE Symposium on Security and Privacy (S&P)*, 2004.

[66] X. Shu, D. D. Yao, and N. Ramakrishnan, "Unearthing stealthy program attacks buried in extremely long execution paths," in *Proceedings of the 22nd ACM conference on Computer and Communications Security*, 2015.

[67] R. Mosli, R. Li, B. Yuan, and Y. Pan, "Automated malware detection using artifacts in forensic memory images," in *Proceedings of IEEE Symposium on Technologies for Homeland Security (HST)*, 2016.

[68] T. Teller and A. Hayon, "Enhancing automated malware analysis machines with memory analysis," 2014.

[69] M. Sharif, V. Yegneswaran, H. Saidi, P. Porras, and W. Lee, "Eureka: A framework for enabling static malware analysis," in *Proceedings of European Symposium on Research in Computer Security (ESORICS)*, 2008.

[70] A. Moser, C. Kruegel, and E. Kirda, "Limits of static analysis for malware detection," in *Proceedings of the 32nd Annual Computer Security Applications Conference (AC-SAC)*, 2007.

[71] 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, February 2012.

[72] A. Moser, C. Kruegel, and E. Kirda, "Exploring multiple paths for malware analysis," in *Proceedings of IEEE Symposium on Security and Privacy (S&P)*, 2007.

[73] 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 *Proceedings of the 18th ACM conference on Computer and Communications Security (CCS)*. ACM, 2011, pp. 363–374.

[74] Y. Fu and Z. Lin, "Space traveling across VM: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection," in *Proceedings of IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2012, pp. 586–600.

[75] M. I. Sharif, W. Lee, W. Cui, and A. Lanzi, "Secure in-VM monitoring using hardware virtualization," in *Proceedings of the 16th ACM conference on Computer and Communications Security (CCS)*. ACM, 2009, pp. 477–487.

[76] J.-Y. Hwang, S.-B. Suh, S.-K. Heo, C.-J. Park, J.-M. Ryu, S.-Y. Park, and C.-R. Kim, "Xen on arm: System virtualization using xen hypervisor for arm-based secure mobile phones," in *2008 5th IEEE Consumer Communications and Networking Conference*. IEEE, 2008, pp. 257–261.

[77] K. Barr, P. Bungale, S. Deasy, V. Gyuris, P. Hung, C. Newell, H. Tuch, and B. Zoppis, "The VMware mobile virtualization platform: is that a hypervisor in your pocket?" *ACM SIGOPS Operating Systems Review*, vol. 44, no. 4, pp. 124–135, 2010.

[78] A. Hayon and T. Teller, "Enhancing malware analysis with automated memory analysis," in *Black Hat Conference, USA*, 2014.

[79] 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*. IEEE, 2012, pp. 549–556.

[80] B. D. Payne, D. d. A. Martim, and W. Lee, "Secure and flexible monitoring of virtual machines," in *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*. IEEE, 2007, pp. 385–397.

[81] greg5678. (2017) Live dangerous linux malware samples! [Online]. Available: https://github.com/greg5678/Malware-Samples

[82] T. Frassetto, D. Gens, C. Liebchen, and A.-R. Sadeghi, "JITGuard: Hardening just-in-time compilers with sgx," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 2405–2419.

[83] Microsoft. (2015) ChakraCore. [Online]. Available: https://github.com/Microsoft/ChakraCore

[84] Theori. (2016) Chakra jit cfg bypass. [Online]. Available: http://theori.io/research/chakra-jit-cfg-bypass

[85] DyninstAPI: Tools for binary instrumentation, analysis, and modification. [Online]. Available: https://github.com/dyninst/dyninst

[86] Super PI single threaded benchmark. [Online]. Available: http://www.superpi.net/

[87] J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, S. Hao, C. Kruegel, and G. Vigna, "Difuze: Interface aware fuzzing for kernel drivers," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2123–2138.

[88] D. Song, F. Hetzelt, D. Das, C. Spensky, Y. Na, S. Volckaert, G. Vigna, C. Kruegel, J.-P. Seifert, and M. Franz, "PeriScope: An effective probing and fuzzing framework for the hardware-os boundary." in *NDSS*, 2019.

[89] W. Wu, Y. Chen, J. Xu, X. Xing, X. Gong, and W. Zou, "FUZE: Towards facilitating exploit generation for kernel use-after-free vulnerabilities," in *27th USENIX Security Symposium*, 2018, pp. 781–797.

[90] The kernel address sanitizer (KASAN). [Online]. Available: https://github.com/google/kasan/wiki

[91] L. W. McVoy, C. Staelin *et al.*, "lmbench: Portable tools for performance analysis." in *USENIX annual technical conference*. San Diego, CA, USA, 1996, pp. 279–294.

[92] E. Cozzi, M. Graziano, Y. Fratantonio, and D. Balzarotti, "Understanding linux malware," in *2018 IEEE Symposium on Security and Privacy (S & P)*. IEEE, 2018, pp. 161–175.

[93] H. Cho, J. Lim, H. Kim, and J. H. Yi, "Anti-debugging scheme for protecting mobile apps on android platform," *the Journal of Supercomputing*, vol. 72, no. 1, pp. 232–246, 2016.

[94] S. Cesare, "Linux anti-debugging techniques, fooling the debugger (1999)."

[95] M. Schallner, "Beginners guide to basic linux anti anti debugging techniques," *Code Breakers Magazine*, vol. 1, 2006.

[96] The Ultimate Packer for eXecutables. [Online]. Available: https://upx.github.io/

[97] 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*. IEEE, 2015, pp. 659–673.

[98] S. Y. Kim, S. Lee, I. Yun, W. Xu, B. Lee, Y. Yun, and T. Kim, "Cab-fuzz: Practical concolic testing techniques for COTS operating systems," in *2017 USENIX Annual Technical Conference (USENIX ATC)*, 2017, pp. 689–701.

[99] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: A platform for in-vivo multi-path analysis of software systems," *ACM Sigplan Notices*, vol. 46, no. 3, pp. 265–278, 2011.