A Novel Dynamic Analysis Infrastructure to Instrument Untrusted Execution Flow Across User-Kernel Spaces

Jiaqi Hong
School of Information Systems
Singapore Management University
jqhong.2015@phdcs.smu.edu.sg

Xuhua Ding
School of Information Systems
Singapore Management University
xhding@smu.edu.sg

Abstract—**Code instrumentation and hardware based event trapping are two primary approaches used in dynamic malware analysis systems. In this paper, we propose a new approach called Execution Flow Instrumentation (EFI) where the analyzer execution flow is interleaved with the target flow in user- and kernel-mode, at junctures flexibly chosen by the analyzer at runtime. We also propose OASIS as the system infrastructure to realize EFI with virtues of the current two approaches, however without their drawbacks. Despite being securely and transparently isolated from the target, the analyzer introspects and controls it in the same native way as instrumentation code. We have implemented a prototype of OASIS and rigorously evaluated it with various experiments including performance and anti-analysis benchmark tests. We have also conducted two EFI case studies. The first is a cross-space control flow tracer and the second includes two EFI tools working in tandem with Google Syzkaller. One tool makes a dynamic postmortem analysis according to a kernel crash report; and the other explores the behavior of a malicious kernel space device driver which evades Syzkaller logging. The studies show that EFI analyzers are well-suited for fine-grained on-demand dynamic analysis upon a malicious thread in user or kernel mode. It is easy to develop agile EFI tools as they are user-space programs.**

I. Introduction

Code instrumentation is used in a myriad of software analysis applications [1], [2], [3], [4]. Its hallmark is to integrate the analysis code and the target code into one binary either before or in the midst of execution. The analysis code executes in the same virtual memory and CPU context as the target, a feature we denote as *native-access* in this paper. Hence, it seamlessly introspects and controls the target by directly accessing the latter's registers and referencing its memory objects with their virtual addresses.

However, recent research [5], [6], [7], [8], [9], [10] 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 [8], [10] are proposed to improve instrumentation transparency and security, they do not tackle the problem at its root and cannot cope with kernel analysis at all. The approach of using the instrumentation code to protect itself faces the cyclic-dependence challenge and results in a complexly-engineered bulky program with a heavy performance toll. In addition to the security and transparency issues, code instrumentation may result in intelligence distortion due to target address space alternation and binary rewriting.

The hardware based event-trapping approach [11], [12], [13], [14] is immune to the aforementioned issues. Often assisted by special hardware facility such as Intel's Performance Monitoring Units (PMU) and ARM debug facility, the approach induces hardware events during target execution. The event is trapped to a more privileged and isolated environment, e.g., the x86 VMX root mode [11], [12], the System Management Mode (SMM) [13] and the ARM Secure World [14]. As these environments are not accessible to the kernel, this approach is often applied for secure and transparent kernel analysis. However, the security strength is attained at the price of native-access. An analysis agent cannot natively read, write, or execute in the target virtual memory as it runs in the isolated environments with an independent address space and a different CPU setting. To bridge the 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 [15], transient attacks [16], [17] and race condition attacks [18], [19]. Hence, the strength and weakness of the event-trapping approach are the opposite of code instrumentation.

In this paper, we propose a new dynamic analysis approach named as **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, EFI shares the virtues of two existing approaches without their drawbacks. We propose and design a virtualization based system infrastructure named as **OASIS** to realize EFI. OASIS empowers the analyzer not only to make native read/write/execute accesses to the target with transparency and isolation, but also to proactively control the target execution for monitoring and tracing purposes. The flow transitions between the target and the analyzer do not precipitate CPU privilege/mode changes. These advantages are attributed to a pivotal design feature: the target's address space is shared with the analyzer, but not vice versa. In comparison, the target and the analyzer share no address space in event-trapping systems [11], [12], [13], [14] and share the space entirely and mutually in code-instrumentation systems [1], [2].

We have built the OASIS prototype on an x86-64 platform as well as three EFI analyzers in two case studies. In the first case study, an EFI analyzer traces full-space control flow transfers including asynchronous executions such as page fault handling. The second case study involves two analyzers working with Google Syzkaller [20]. One makes postmortem analysis of a kernel crash report and the other explores a malicious kernel driver with fuzzed inputs. Despite analyzing the kernel, both analyzers are developed and launched as user space programs, with the capability to control and introspect the target. The case studies show that EFI applications are handy and agile and well-suited to fine-grained and on-demand analysis on (malicious) kernel threads. We have also rigorously evaluated OASIS performance and security with benchmarks and experiments. OASIS and EFI applications remain transparent and effective against targets armed with anti-analysis techniques including packing.

CAVEAT. We acknowledge that detection of virtualization remains possible. Nevertheless, the features of hardware assisted virtualization are so appealing that commodity operating systems, e.g., Microsoft Windows 10 have started to adopt them. Hence, the presence of virtualization and a hypervisor alone is not decisive enough for future malware to inhibit its malevolent behaviors.

Organization. Next, we briefly review related work. We then present an overview of OASIS and EFI in §III, and explain the details of analyzer execution and target execution in §IV and §V, respectively. EFI techniques are elaborated in §VI followed by its security and transparency analysis in §VII. We present two case studies in §VIII and report experimental results on performance and security evaluation in §IX. Finally, we discuss potential extensions and conclude the paper in §X.

II. Related Work

Code instrumentation (including binary rewriting/translation in a general sense) has been widely used in malware analysis. Static binary instrumentation is applied to collect malware's runtime information regarding system calls, API calls [21] or even control flow. Dynamic binary instrumentation (DBI) modifies the target during its execution. Intel Pin [22], Valgrind [23] and Dyninst [24] are well-known versatile DBI frameworks. TaintCheck [25] uses Valgrind to make dynamic taint analysis. Instead of using just-in-time compilation and emulators, LiteInst [3] proposes to use instruction punning to insert probes into running code. More details can be found in DBI surveys [26], [10]. However, recent results [5], [6], [7], [8], [9], [10] show that sophisticated malware may detect DBI existence and therefore inhibits the malicious behaviors. This limitation is one of the motivations of our work.

Our work is closely related to kernel analysis. It is more challenging than user space program analysis since the kernel is the system software managing the platform. DBI has been applied to kernel analysis as well. PinOS [2] extends Pin [22] to instrument the Linux kernel. Cobra [1] uses DBI to analyze obfuscated code in both user- and kernel-mode. The other approach to kernel analysis is to leverage hardware features to trap the kernel execution into a more privileged system software. One of the hardware features is CPU and MMU virtualization. Gateway [27] uses a hypervisor to monitor how a driver invokes kernel API. Ether [11] is the first hypervisor-based framework for kernel analysis with the emphasis on transparency. SPIDER [12] achieves stealthy instruction-level tracing by using an instruction probe (i.e., INT3) to trap to the hypervisor. Intel SMM is used in MALT [13] to attain stronger transparency than virtualization offers. Ninja [14] holistically explores a set of ARM features including TrustZone, PMU and Embedded Trace Macrocell (ETM), for cross-space debugging and analysis. The side-effect of the trapping approach is that the analyzer cannot natively access the target, which is also one motivation of our work.

Another related area is virtual machine introspection, especially out-of-VM introspection such as process out-grafting [28], Virtuso [29], VM space-traveling [30], Hybrid-bridge [31], and ImEE [19]. 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 [32]) 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 our work. It tackles the semantic-gap problem [18] 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 natively read the target virtual memory. Nonetheless, the agent does not have its own address space and cannot control the target. Hence, ImEE can only be used as a memory introspection engine, instead of a tool for comprehensive dynamic analysis.

III. Synopsis

A. System & Adversary Models

We consider a commodity x86-64 multicore platform with CPU and MMU virtualization. The platform is managed by a host OS (e.g., Linux KVM) running on cores in the VMX root mode. In the rest of the paper, we also refer to the host OS as "the hypervisor" when the functionality in use is related to virtualization. The *target* is a running thread in the hardware-assisted virtual machine denoted as the *guest*. 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. It is orthogonal to our study to cope with memory corruption attacks [33] exploiting vulnerabilities in the analyzer's implementation.

B. High-level Problems

To realize execution flow instrumentation, we need to tackle three high-level problems. The first and foremost is how for the analyzer to natively and securely control the target and introspect its virtual address space. Code instrumentation introspects and controls the target natively, but not securely. Although ImEE [19] proposed a method for an agent to introspect the target natively and securely, the agent cannot control the target, e.g., to change the target control flow. The event-trapping approach can securely control the target, but cannot make native introspection.

The second problem is how to securely and efficiently interleave the analyzer instruction flow with the target flow in an on-demand fashion. As discussed in §I, it is not secure to mix up their instructions in one address space. The event-trapping approach requires a hardware event to trigger the switches. Besides its heavy performance toll due to the events, this approach lacks of agility as the control strategy is constrained by the hardware support.

The third problem is how for the analyzer to receive its host OS services including system calls and I/O operations. The significance of solving the problem is that the analyzer can be developed as a regular user space program, which simplifies its implementation and deployment. Schemes in [11], [12], [13] inevitably require programming in a more privileged space than the kernel (e.g., in the hypervisor). The agent in ImEE [19] cannot issue system calls and does not have its own address space.

Our solution involves low-level details of address translation with virtualization in x86-64 platforms. The technical background can be found in Appendix A.

C. Overview of OASIS

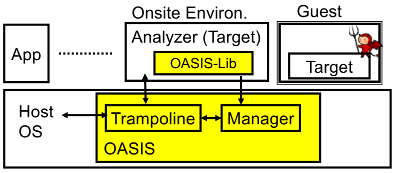


Fig. 1. An architectural view of OASIS based analysis with an untrusted guest including its kernel. The yellow boxes highlight OASIS software composition.

We propose OASIS as the infrastructure for EFI. Its architecture (in Figure 1) consists of three software components: the Manager, the Trampoline and OASIS-Lib. The Manager sets up and manages the *onsite environment* which is the system environment exclusively for EFI. The 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 (including OASIS-Lib) and the target are the only software running in the onsite environment, with all their physical pages in the host and the guest, respectively. As a regular application in the host, the analyzer is loaded into the environment immediately after process creation, while the target thread in the guest is captured and migrated to it. The analyzer instruments the target execution flow inside the onsite environment.

- When the analyzer runs, the onsite environment uses A-EPT and O-PML4 to construct the *analyzer/target paging hierarchy* that merges the target space with the analyzer. The Trampoline handles the analyzer's interactions with the host OS so that the analyzer's system calls and events (e.g., page faults) are served properly. (Details in Section IV)
- When the target thread runs, the onsite environment uses T-EPT, O-PML4, O-PDPT, O-PD and O-PT to construct the *target/lib paging hierarchy* that merges the target space with OASIS-Lib. All the target's memory accesses are still upon its physical pages in the guest. (Details in Section V)

As a library to the analyzer, OASIS-Lib consists of one code page and several data pages to facilitate 1) control transfers between the target and the analyzer; and 2) event handling in the onsite environment. OASIS's high-level approaches to the three aforementioned problems are sketched below.

Native Target Access With Isolation. The analyzer/target hierarchy synthesizes the analyzer's paging structures in the host with the target's in the guest. It defines a hybrid virtual address space so that the analyzer instructions natively access *both* component spaces with ensured mapping consistency. The target/lib hierarchy appears the same to the target as in the guest although it encloses secret mappings for OASIS-Lib. It does not have any mapping to the analyzer space, effectively blocking target accesses to the analyzer. The analyzer shares the onsite CPU with the target so that it can control the latter's execution.

Instruction Flow Interleaving. Instruction flow interleaving is realized by switching between the two hybrid paging hierarchies. No interrupt or exception is triggered. The switches are carried out by two short instruction segments in OASIS-Lib. The one switching from the target to the analyzer is the *exit-gate*, and the other is the *entry-gate*. A combination of virtualization, software and randomization techniques are used to achieve their functionality, security and transparency.

Transparent OS Services. The analyzer/target paging hierarchy paves the way for transparent OS services as the host kernel's view of the analyzer's address space remains valid and unchanged. Since the onsite core is in the non-root mode, system calls and events during the analyzer execution are trapped to OASIS which hands them over to the host OS as if they originate in the analyzer running as a host application.

D. Overview of OASIS Based EFI

The high level workflow of dynamic analysis using EFI is illustrated in Figure 2. To start the analysis, OASIS exports the target thread 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.

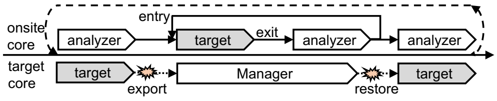
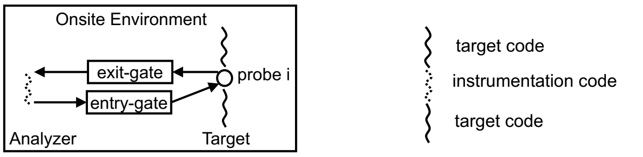


Fig. 2. The high-level workflow of EFI analysis. It consists of cyclic switches between the target and the analyzer within the onsite environment.

The analyzer uses *probes* to specify the instrumentation junctures. When the target execution flow reaches a probe, it transfers the control 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 nor affects other guest threads sharing the target code. OASIS ensures probe transparency and security. Figure 3(a) depicts the i-th round interleaving in an EFI session which has the same effect as code instrumentation in Figure 3(b), but with stronger security and transparency.



(a) Interleaving execution flows w/o CPU privilege or mode changes

(b) Equivalent effect achieved by using code instrumentation

Fig. 3. Illustration of execution flow instrumentation via probes and gates.

IV. Analyzer Execution in Onsite Environment

A. The Analyzer/Target Hierarchy

Loaded to the onsite environment for the analyzer execution, the analyzer/target hierarchy maps the analyzer's and the target's virtual addresses, so that the analyzer natively accesses both spaces. We first explain below the split of 48-bit VA and 48-bit GPA spaces before describing how to construct the hierarchy. Note that 48-bit space consists of 512 512-GB bands.

1) Virtual Addresses: The host OS is modified to allocate the analyzer's all VA regions in one 512-GB band with a 512-GB aligned base in the lower half of the 48-bit space. As a result, the analyzer's paging hierarchy has only one PDPT page. When the analyzer is loaded to the onsite environment with the analyzer/target hierarchy, OASIS chooses one 512-GB band unoccupied by the target kernel as its new VA region. (In Section VII we explain how to cope with situations where no free 512-GB band is available.) The analyzer VA in the host and its counterpart in the hybrid space differ in the leading 9 bits only. Since the analyzer never runs in the host, its new VA regions are transparent to the host. OASIS-Lib is also loaded in the 512-GB band in the target/lib hierarchy. Moreover, OASIS reserves the first page and the last page of the chosen band for no mapping. These two pages serve as the "air gap" between the analyzer and the target to counter an attack described in Section VII.

2) Guest Physical Addresses (GPAs): When the analyzer runs in the onsite environment, the analyzer/target paging hierarchy translates VAs to GPAs which are further translated to host physical addresses (HPAs) by A-EPT. It is also vital to separate the analyzer and the target's GPA ranges in A-EPT. The guest's GPA domain is denoted as $[0, T]$ where T is determined according to the guest configuration. Suppose that PA_{max} is the maximum physical address of the platform, the analyzer GPAs are assigned between T and PA_{max} so that they do not overlap with the guest.

3) Details of Address Mappings: The idea of constructing the analyzer/target hierarchy is to fuse their spaces by linking their PDPT pages into one PML4 page. The `CR3` of the onsite core is loaded with a random GPA (denoted by α) in $(T, 2^{48})$. All entries on the target's PML4 page in the guest are copied to O-PML4. Suppose that the λ-th 512-GB band is unused by the target kernel and is chosen for the analyzer. OASIS populates the λ-th entry in O-PML4 with another random GPA (denoted by β) in $(T, 2^{48})$ and clears its supervisor bit. A-EPT is configured to map four types of GPAs as below.

- GPA α is mapped to the HPA of O-PML4 so that O-PML4 is the root of the analyzer/target hierarchy.
- For all GPAs used in the guest, their GPA-to-HPA mappings are the same as in the guest and all execution permissions are removed. These mappings graft the target's paging hierarchy to the analyzer/target hierarchy since its PML4 page and O-PML4 have the same entries except the λ-th one. They also ensure the target VAs are eventually mapped to the same HPA as in the guest.
- GPA β is mapped to the HPA of the analyzer's PDPT page in the host so that the paging hierarchy managed by the host OS for the analyzer (except the PML4 page) is grafted to the analyzer/target hierarchy. Note that the analyzer's hierarchy has only one PDPT page.
- 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. Since the analyzer GPAs are required to be larger than T, the host OS is customized to satisfy it.

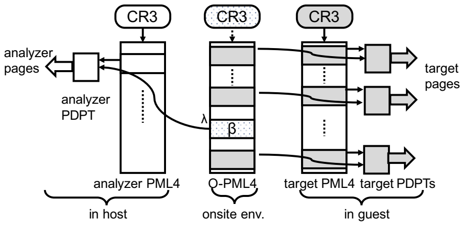


Fig. 4. The analyzer/target hierarchy is rooted at O-PML4 whose λ-th entry grafts the analyzer's hierarchy in the host and other entries graft the target's hierarchy in the guest. The boxes with patterns indicate randomized values.

The resulting hierarchy is visualized in Figure 4. All target VAs are mapped to the corresponding physical pages in the same way (except permissions) as in the guest. Hence, the analyzer instructions can natively access the target virtual memory. All analyzer VAs are also mapped to the corresponding physical pages as in the host, 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) automatically take the same effect in the onsite environment. To ensure consistency between the target's PML4 and O-PML4, OASIS sets the former as read-only in the guest EPT. The guest kernel's PML4 modification (which is rare in 64-bit platforms) is then intercepted and cloned to O-PML4. An update on the analyzer's mappings by the host OS also takes an immediate effect, except when a new physical page is allocated. For instance, a page is allocated for the analyzer heap expansion or a new paging structure page is added to the paging hierarchy. These situations require A-EPT to create new mappings. As described shortly, it is dealt with after the host OS completes its system call or exception service for the analyzer.

B. Analyzer Execution

The analyzer's system calls and events are still handled by the host OS. OASIS installs new IDT, GDT, TSS and a stub exception handler for the analyzer execution. Although the analyzer runs in Ring 3 initially, it may enter Ring 0 by either inheriting the target's privilege or calling the special call gate in the GDT. Figure 5 depicts how events during analyzer execution are handled by the host OS with the details elaborated below.

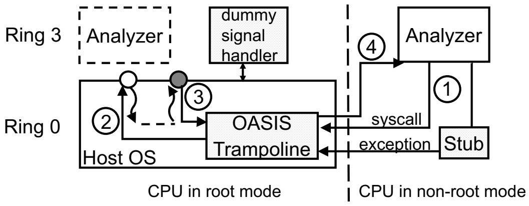


Fig. 5. Event handling workflow for the analyzer's execution. The white circle denotes the entry of the kernel event handler while the dark circle denotes the user space entry for the kernel to return. The dash line refers to the kernel handler execution.

1) Analyzer-to-Host Switch: The analyzer system calls and exceptions trigger a VM-exit to the Trampoline (Step 1) in different ways. A system call triggers an EPT violation since the handler located by the hardware is not executable under A-EPT. An exception (e.g., a page fault) is trapped to the stub which saves the context and issues a hypercall. In both scenarios, the Trampoline prepares the context for the host kernel handler according to the vCPU state in the Virtual Machine Control Structure (VMCS) and the context saved by the stub. It also prepares the host kernel stack and loads relevant registers, according to the hardware behavior in handling exception.

Next, the Trampoline invokes the corresponding host OS handler in a "returning-boomerang" fashion, in the sense that it regains the control after the service is completed. Specifically, it sets a local hardware breakpoint at the kernel's return to the analyzer, and then *jumps* to the entry of the kernel handler (Step 2). Unless the analyzer thread is aborted or killed, the host kernel is expected to return to user-mode. The breakpoint hijacks the control flow right after `iret` or `sysret` so that the Trampoline regains the control (Step 3). From the kernel perspective, the thread has been returned to user-mode. Hence, the hijacking does not cause inconsistency in kernel states.

Signal delivery to the analyzer requires a special treatment since the kernel calls the analyzer's own signal handler if any. Since the host kernel is oblivious to the analyzer execution in the onsite environment, it cannot invoke the analyzer handler. To preserve the semantic of signals, a dummy user-space signal handler is registered to the host OS on behalf of the analyzer. When invoked by the OS, the dummy handler saves the parameters (if any) and sets a flag to indicate its invocation. The analyzer's handler is then invoked after it resumes in the onsite environment.

2) Host-to-Analyzer Switch: When the Trampoline regains the control, it removes the breakpoint and calls the Manager to make proper preparation for the onsite environment reentry. If the host OS allocates new physical pages to the analyzer's paging hierarchy or the analyzer's space, the Manager updates A-EPT to create the corresponding GPA-to-HPA mappings. For a system call return, the Manager updates the saved vCPU's `RIP`, `RAX`, `RFLAGS` according to the host handler's returned context `RCX`, `RAX`, and `R11` respectively. For the system call `arch_prctl`, the Manager updates the vCPU's `FS` base accordingly. In the end, the Trampoline switches the CPU to the VMX non-root mode using the saved VMCS so that the analyzer resumes (Step 4). Note that the procedure to launch the analyzer in the onsite environment for the first time is similar to the one above. The main difference is that the starting instruction in loading is the entry of the host's loader.

3) VA Adjustment: Since the analyzer's 512-GB band in the onsite environment is different from the one in the host, VA related data exchanged between the analyzer and the host OS handler must be adjusted accordingly. During VM-exit due to system calls, interrupts and exceptions, the Trampoline updates all VAs passed to the host OS, e.g., addresses in system call parameters held by registers and the faulting instruction address in `CR2`. When re-entering the onsite environment, it updates VAs returned to the analyzer, such as the new program break from a `brk` system call.

V. Target Execution in Onsite Environment

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

A. Address Mapping For Target Execution

The target/lib hierarchy merges the target address space and OASIS-Lib's. The approach is the same as in the analyzer-target hierarchy, except that OASIS-Lib's hierarchy is built by OASIS using O-PDPT, O-PD and O-PT. It uses the same `CR3` and O-PML4 as in the analyzer/target hierarchy. T-EPT clones all GPA-to-HPA mappings in the guest including their permissions so that all target VAs and GPAs are translated in exactly the same way as in the guest. T-EPT maps GPA β to O-PDPT's HPA. One randomly chosen O-PDPT entry is populated with a random GPA mapped to O-PD by T-EPT. Similarly, one random O-PD entry is filled with a random GPA mapped to O-PT. Six consecutive O-PT entries are randomly chosen and assigned with random GPAs mapped to OASIS-Lib HPAs. As a result, OASIS-Lib is in a random region within the analyzer's 512 GB band. Figure 6 visualizes the target/lib hierarchy. Note that *no* analyzer page is mapped under T-EPT.

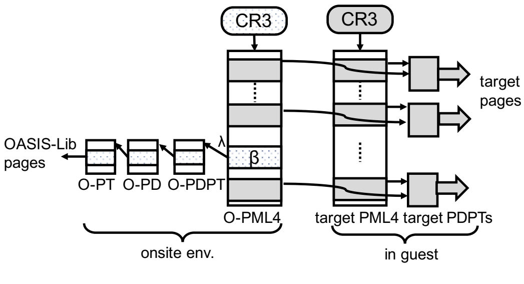


Fig. 6. The target/lib hierarchy rooted at O-PML4 using GPA-to-HPA mappings in T-EPT. The boxes with patterns have random GPAs.

When the guest EPT is updated by the host OS, OASIS synchronizes it with T-EPT and A-EPT so that both of them have the same GPA-to-HPA mappings as used in the guest.

B. Execution Consistency

The target/lib hierarchy ensures consistent memory references to the target memory in the guest. We explain how to handle the CPU context as well as interrupts and exceptions.

Exportation & Restoration. The target thread is captured during 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 held by OASIS until the target thread is restored so as to mimic the target's CPU occupancy in the guest and to 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 its onsite environment context.

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 underlying 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 delivered to QEMU which notifies the hypervisor to inject the interrupt into vCPU during VM-entry. In OASIS, the idea is to use the Manager as the proxy to make I/O operations on behalf the target. The target's I/O operation in the onsite core is trapped to the Trampoline and forwarded to the Manager holding the target core (shown in Figure 2). The Manager executes the operation so that it appears to the host OS as a request from the target core. As a result, the host OS passes it to the QEMU process supporting the guest VM. After I/O 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. Thus, I/O operations in the onsite environment appear the same to the target as in the guest.

System Data Structure Relocation. To support analysis on the target's exception and interrupt handling, OASIS relocates the target's IDT, GDT, and TSS to OASIS-Lib. If needed, the analyzer can customize these relocated data structures to monitor and control asynchronous events in the exported target execution. For instance, an analyzer monitoring the target's page fault handling hooks the target's INT#14 hander to capture the event.

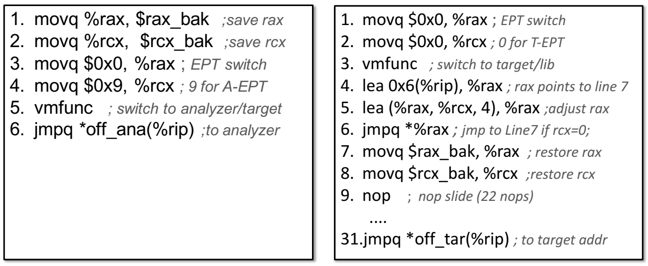
To protect transparency, OASIS prevents the target from accessing the relevant registers of the onsite core by configuring the VMCS structure. Any software access to them is trapped to the analyzer which returns the original content. The target thread can still read these tables at their original VAs, because they remain in the guest with no modifications. Updates to the tables in the guest are intercepted so that OASIS clones the changes to the relocated counterparts.

CAVEAT. The design above is only for target execution in the onsite core. It has no effect on and is transparent to threads running in the guest.

C. Cross-Flow Control Transfer

The control flow of the target can be transferred to and from the analyzer through the exit-gate and the entry-gate, which is the embodiment of EFI. The cross-flow control transfers are realized by switching between T-EPT and A-EPT, which essentially switches between the target/lib and analyzer/target paging hierarchies. We use the `vmfunc` instruction[1] to switch the EPTs. The instruction following `vmfunc` is fetched from the new hierarchy.

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 mapped as writable under A-EPT 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.



```
1. movq %rax, $rax_bak   ;save rax
2. movq %rcx,  $rcx_bak  ;save rcx
3. movq $0x0, %rax ; EPT switch
4. movq $0x9, %rcx ; 9 for A-EPT
5. vmfunc   ; switch to analyzer/target
6. jmpq *off_ana(%rip) ;to analyzer
```

(a) Exit-gate

```
1. movq $0x0, %rax ; EPT switch
2. movq $0x0, %rcx ; 0 for T-EPT
3. vmfunc   ; switch to target/lib
4. lea 0x6(%rip), %rax ; rax points to line 7
5. lea (%rax, %rcx, 4), %rax ;adjust rax
6. jmpq *%rax ; jmp to Line7 if rcx=0;
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

Fig. 7. Assembly code of the exit-gate that passes the control to the analyzer and the entry-gate that returns the control to the target.

Exit-gate. Figure 7(a) presents the assembly code of the exit-gate which runs in 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. It then issues `vmfunc` with parameters instructing the hardware to switch to A-EPT. Finally, it jumps to the analyzer's handler whose address is stored in a data page mapped read-only in T-EPT and read-writable in A-TEPT. Note that the jump instruction is fetched and executed from the analyzer/target hierarchy under A-EPT. The OASIS-Lib code page is mapped as executable in both A-EPT and T-EPT. To minimize the code size, the exit-gate does not save the target's CPU context except two registers used by itself. The analyzer inherits the target CPU context and retrieves the target's `RAX` and `RCX` from the data page.

Entry-gate. Figure 7(b) presents the assembly code of the entry-gate which runs in the analyzer flow to pass the control to the target. It issues `vmfunc` with parameters instructing the hardware to switch to T-EPT. Right after the switch, Line 4 to 6 check whether `RCX` is indeed 0 indicating a switch to T-ETP. If so, it jumps to Line 7 to restore `RAX` and `RCX`; otherwise it jumps to Line 31. The code is crafted in this way to do the checking without affecting `EFLAGS`. The destination

[1] According to Intel specification, when `RAX` is 0, `vmfunc` loads the EPT priorly prepared the hypervisor according to an index value stored in `RCX`. No VM-exit is incurred during the EPT switch.

of the final jump is specified by the analyzer by placing the destination in a read-only page under T-EPT. 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 (supposedly) fetched from the target/lib hierarchy. A slide of 22 `nop` instructions is placed before the final `jmp` instruction. The slide is long enough to accommodate two instructions for make-up execution due to the probe (as explained in §VI-B).

In short, the cross-flow control transfers do not incur any CPU privilege or mode changes. As shown in Figure 8, OASIS-Lib is mapped in both hierarchies wherein the analyzer and the target use the *same* VAs for IDT, GDT and TSS, but different sets of physical pages. The analyzer does not need to adjust the corresponding registers in a cross-flow switch.

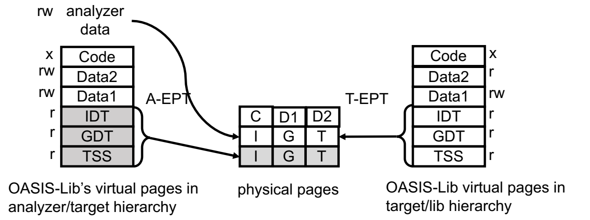


Fig. 8. OASIS-Lib is mapped in the same VA region in both hierarchies. The shadowed physical pages are for analyzer execution only. The analyzer accesses the target's IDT, GDT and TSS from its own data section. D1 is used in the exit-gate to save registers while D2 is used in both gates to load the final jump destinations.

VI. Execution Flow Instrumentation

We have described how the analyzer and the target run in the onsite environment and the mechanism to interleave the flows. In this section, we explain how the analyzer flexibly and securely specifies the junctures for interleaving. The basic idea is to use *probes* [3], [10] to replace target instructions at the desirable virtual addresses. When the target flow reaches a probe, it is directed to the exit-gate. The analyzer can install and remove the probes at any time during target execution.

A. 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.

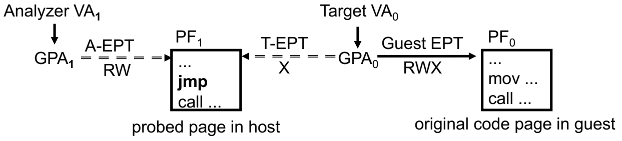


Fig. 9. EPT redirection on T-EPT for probe installation and uninstallation.

The mechanism is illustrated in Figure 9. Suppose that target code page VA$_0$ is mapped to physical page frame PF$_0$ in the guest via GPA$_0$, and that the analyzer has a data page VA$_1$ mapped to physical page frame PF$_1$ in the host. To install a probe in VA$_0$, the analyzer copies it to VA$_1$, inserts the probe to code in VA$_1$, and then issues the page-substitution hypercall. In response, OASIS modifies T-EPT to map GPA$_0$ to PF$_1$ 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$_1$ instead of PF$_0$. The analyzer running under A-EPT can read/write the probed page PF$_1$ via VA$_1$. When the probe in VA$_0$ is no longer needed, the analyzer uses the page-reinstatement hypercall so that GPA$_0$ is mapped back to PF$_0$.

Permission Conflict Resolving. For the sake of transparency, we follow the approach in SPIDER [12] to redirect the target's read and write to VA$_0$ back to PF$_0$. The basic idea is to load the onsite core's data TLB with EPT mappings to PF$_0$ (with read-only permission) and to load the instruction TLB with EPT mappings to PF$_1$ (with execution-only permissions). The target's write to VA$_0$ is single-stepped so that a modification on PF$_0$ is also cloned to PF$_1$. The key difference between OASIS and SPIDER [12] 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$_0^*$ for a malicious purpose or out of benign reasons such as page swapping. Although the change does not affect the target execution in the onsite environment, it invalidates the probe in VA$_0$ unless GPA$_0^*$ is also properly mapped to in T-EPT. 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$_1$.

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 it runs in a different core from the target core, it is expected to notify the target thread to invalidate the TLB at the (supposed) target core so as to avoid mapping inconsistence. The cross-core notification is captured by the OASIS Manager which then updates T-EPT accordingly.

B. EFI Probes

An EFI probe kickstarts the transition from the target flow to the analyzer via the exit-gate. Depending on 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*. The INT3-probe, as used in [3], [12], 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 to OASIS-Lib 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 probed page can be more than 2GB, which is beyond the maximum range of any addressing mode without using a general register. However, involving a register in the probe mandates saving its content to the target memory, which is intrusive and undermines transparency.

Our approach is to jump to the exit-gate via a call gate whose selector can be easily formed using bytes in pages near to the probe. In specific, the analyzer initially populates all unused entries in the *target* GDT with call gates whose Descriptor Privilege Levels are 3 and destinations are the exit-gate. The JMP-probe is a far jump instruction to one of the installed call gate, in the form of:

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

where the sum of `RIP` and `offset` addresses a call gate selector in the memory. When installing the JMP-probe, the analyzer composes 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 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 are as follows. The analyzer randomly picks 16 bits from one target code page with the only restriction that the 3rd least significant bit be '0', which implies a GDT segment selector instead of a LDT selector. It then checks whether bits 3 to 15 match any call gate entry. Since most of the 8K GDT entries are the call gates, the chance of succeeding is significantly high[2]. If it fails, the analyzer picks and checks another 16 bits. Once the qualified 16 bits are 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 a new one 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 well-known thorny issue about probes is the make-up execution of the affected target instruction(s) after handling the probe. Two solutions are used in the literature. One is to emulate those instructions in a separate space and the other is to single-step them after restoration and then replace them with the probe again [12]. Obviously, both solutions are cumbersome and incur significant CPU time.

It is comparatively easier and faster to resolve the issue in OASIS because, benefiting from native-access to the target, the analyzer does not need to emulate the target execution or single-step it. For the JMP-probe, the analyzer writes back the replaced target instruction(s) and resumes the target execution from the probed VA. 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 accordingly if it is a call instruction. For a non-transfer instruction affected by the INT3-probe, the analyzer copies it to the entry-gate's NOP-slide and rewrites it according to the new VA. If the memory operand in the original instruction is referenced using RIP-relative addressing, it is revised to use register-indirect addressing mode followed by another instruction to restore the register used in addressing. The total length of the two make-up 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.

C. 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 with a fine-grained and adaptive control over the target execution in user and kernel modes.

1) EFI With Tracing: To trace the target, the JMP-probe is installed and then removed (after being executed) along a sequence of virtual addresses in the target instruction flow. A requirement for successful tracing is no transfer instruction between two adjacent probed VAs. Hence, the tracing granularity can be single-step, instruction slices and basic blocks.

The probed sites can be decided during preprocessing (e.g., for data flow tracing) or at runtime (e.g., for control flow tracing). When the target flow reaches the probe, the control is switched to the analyzer through the JMP-probe and the exit-gate. After executing its analysis function, the analyzer restores the bytes replaced by the probe, fetches (or determines) the next site and installs the probe there. It then updates the entry-gate so that the target resumes execution from the probed VA at present 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 determines the transfer destination of the instruction and passes the control to it via the entry-gate. Note that it inherits the target CPU context and also has native accesses to the virtual memory.

2) EFI With Breakpoints: The INT3-probe is 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 technique to make up for the affected instruction execution.

Specifically, the analyzer configures the target's relocated GDT, IDT and TSS to provide a new exception stack and a new handler. When an INT#3 exception 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 exception stack according to the guest system setting, and then jumps to the target's own INT#3 handler.

3) 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 (INT#14) as an example as it is often used by the kernel to manage virtual address spaces. The analyzer installs a new INT#14 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.

VII. EFI Security and Transparency Assessment

EFI *security* means that the adversary cannot tamper with the analyzer and OASIS code, data and control flow, while EFI *transparency* means that the adversary cannot detect any analysis-related artifact. We first assess them against the kernel adversary running in the guest. No artifacts of OASIS or the analyzer are exposed to the guest and no modification

[2] Since most Linux kernel uses less than 8 entries in the GDT, the success probability of one checking is $1 - \frac{8}{8 \times 2^{10}} \approx 0.999$.

is made on the guest software or hardware settings. As in other virtual machines, all memory accesses from the guest are restricted to the mapped physical pages. Hence, no direct attack from the guest compromises security and transparency. The adversary may use the page tables in the guest to manipulate the analyzer's access to the target. This attack is equivalent to feeding poisonous data to the analyzer, an indispensable risk for all dynamic analysis systems. Note that side-channel attacks against transparency are possible.

Next, we assess security and transparency against the exported target thread. The onsite core's `IDTR`, `GDTR` and control registers are set as inaccessible to the target. A read access to them is returned with the original content while a write access only updates the saved copy. With T-EPT, the target does not have mappings to physical pages owned by the analyzer or OASIS, except OASIS-Lib. The library consists of one code page (execution-only), four read-only data pages and one writable data page. In addition, four paging structure pages are used in the target/lib hierarchy although no VA is mapped to them. Note that access permissions on OASIS-Lib pages are all set in T-EPT. Below is a detailed analysis regarding OASIS-Lib which shares the VA space with the running target.

A. Security Against the Exported Target

Supposing that the VAs and GPAs used by OASIS-Lib are exposed, we consider attacks modifying OASIS-Lib data or executing its code. The only writable page in OASIS-Lib is used for the exit-gate to save `RAX` and `RCX`. Hence, an illicit modification only implies faked data passed to the analyzer. The only executable code in OASIS-Lib is the exit-gate, the entry-gate, and a handler hook. Unless A-EPT is loaded, any malicious execution only affects the target itself instead of the analyzer or OASIS. Although the target can maliciously load A-EPT to the onsite core, it cannot change the entry to the analyzer. Note that no target page is executable under A-EPT.

According to OASIS configuration, the target must run `vmfunc` with $\texttt{RCX} = 9$ and $\texttt{RAX} = 0$ to load A-EPT. It may use `vmfunc` in its own code. Thus, the instruction next to `vmfunc` must be in the target page, because the analyzer space is sandwiched by the air gap described in §IV. Since no target page is executable under A-EPT, this instruction triggers an EPT violation. Alternatively, the target may misuse `vmfunc` in the exit-gate and the entry-gate by jumping to it. For both gates, the target cannot change the final transfer destination which is chosen by the analyzer and stored in a page the target has no write permission. Since the jump instruction in the exit-gate is next to `vmfunc`, it is ineffective to misuse the exit-gate.

Different from the exit-gate, the entry-gate may contain two target instructions for make-up execution under T-EPT. Hence, it is crafted to prevent them from being executed when the gate is misused to load A-EPT. Specifically, the three instructions following `vmfunc` detect whether `RCX`=0. The first two `lea` instructions prepare the destination of the jump instruction in Line 6 (in Figure 7(b)), in such a way that the control flow jumps to Line 7 if `RCX`=0; or to the final jump instruction in Line 31 if `RCX`=9. (`RCX` cannot be other numbers as there is no corresponding EPT.) Hence, when the entry-gate is misused to load A-EPT, the NOP-slide or the make-up instructions are skipped. The final jump triggers an EPT violation because its destination chosen by the analyzer is not executable under A-EPT. In short, in the event that OASIS-Lib's transparency is broken, the target can only feed the analyzer with faked `RAX` and `RCX`.

B. Transparency Against the Exported Target

When the target runs, all OASIS related artifacts (e.g., the GDT, IDT, `CR3`, and probed pages) are protected against the target accesses by virtualization configuration, except that OASIS-Lib is hidden by randomization. The target may guess OASIS-Lib VA/GPA and then verifies it through read or allocation.

Guess and Read. In the target/lib hierarchy, OASIS-Lib occupies 6 continuous pages with a random base in the analyzer's 512-GB band. Supposing that the band is known, the average probability of a successful guess of one OASIS-Lib page is $\frac{6\times 2^{12}}{2^{39}} \approx 2^{-24}$. The target can also map its own VA to a guessed OASIS-Lib GPA and accesses that VA. OASIS-Lib has 10 GPAs randomly chosen from $(T, 2^{48})$, including those for its 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 success probability of direct accesses is significantly low. A wrong guess leads to an EPT violation and OASIS crashes the target since it is the supposed consequence in the guest.

Guess and Allocate. The exported target may allocate VA regions to create a contention with the analyzer and OASIS-Lib. Since the target has the full access to its entire paging structure pages (including its PML4 page), the allocation always succeeds. The conflict is caught when OASIS synchronizes O-PML4 with the target's modified PML4 in the guest. The VA contention can also appear during initialization wherein all PML4 entries are occupied by the target. Note that it is infeasible to relocate the analyzer at runtime.

Our solution is for the analyzer/target and target/lib hierarchies to use different PML4 pages by mapping `CR3` differently in A-EPT and T-EPT, without aborting the present EFI session. The analyzer/target hierarchy uses a new PML4 page with the same content as before, and the analyzer remains in its 512-GB band. When the analyzer needs to access a target VA (say ν) in the same 512-GB band, OASIS links one priorly reserved entry in the analyzer's PDPT to the target's PD page translating ν. As a result, it can access ν via a relocated VA. The target-lib hierarchy still maps both the target and OASIS-Lib so that VAs in the 512-GB band are mapped in the same way as the target, except those six pages in OASIS-Lib. The hierarchy is constructed in the similar way as in §IV except that O-PML4, O-PDPT, O-PD and O-PT have entries cloned from their guest counterparts (which are monitored by OASIS), except the path translating OASIS-Lib. Since memory allocation does not expose OASIS, the target still needs to guess an address and read. The chance of choosing OASIS-Lib pages remains at 2^{-24}. However, since a failed guess does not trigger an EPT violation, the target may keep guessing. Hence, in average it needs 2^{23} guesses to detect OASIS-Lib. To further strengthen OASIS transparency, the analyzer with a long-lasting session can periodically request to relocate OASIS-Lib to another random location inside the 512-GB band, which is realized by the host OS and OASIS in the same way as OASIS-Lib loading. After relocation, the analyzer re-installs all existing probes in the target if any.

VIII. Case Studies

We have implemented OASIS on a PC with an Intel Core i5-4590 3.3 GHz processor (supporting VT-x) and 16 GB DRAM. The host OS is Linux kernel 3.13.0 with KVM for 64 bit x64 SMP. The guest runs the same Linux version with 4 GB memory. OASIS consists of 4148 SLOC with the majority in the host kernel. We develop three analyzers in two case studies. (See Appendix B for more implementation details.) All use Dyninst [24] APIs (with slight modification for loading memory-resident binaries) to disassemble the target binary code before analysis or on-the-fly to extract instruction level semantics.

A. Case Study I: Full-space Control Flow Tracing

The first case is a full-space EFI control flow tracer using the JMP-probe. It is tested against three Linux shell commands (`ls`, `pwd`, and `kill`) and SuperPI [34] 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 very 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 exits 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 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 I below.

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 I. Control Flow Tracing Report.

The OASIS based EFI tracer has its pros and cons as compared to hardware-aided full-space tracers such as MaLT [13] using PMU and Ninja [14] using ARM ETM. We report the target slowdown in Table II, where "preprocessing" means offline target disassembling. The EFI tracer outperforms MaLT since it triggers no hardware events. Although ARM ETM does not incur overhead for tracing, it only outputs the VAs. Unlike MaLT and OASIS, Ninja cannot control the target, e.g., to suspend the target before data fetching. The EFI tracer is better at introspection and control due to its native-access and CPU mode sharing with the target. Since the hardware facility typically reports the virtual address of the monitored event only, a hardware-aided tracer running in a higher privileged environment must bridge the gap to retrieve data from the target. A more noticeable advantage of our tracer 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 tracer is restricted to the types of events and instructions supported by the facility. It cannot be customized by the analyst to meet different demands. For instance, a data flow tracing can only be achieved with single-stepping if using MaLT. The limitation of the EFI tracer is that all transfer instructions must be checked to avoid losing the control while the hardware-aided tracer can support coarser granularity, e.g., at the function level. We remark that the EFI tracer is better at semantic-driven tracing (e.g., data flow and memory operations) which demands deep analysis and fine-grained control, while the hardware-aided tracer is more suitable for semantic-neutral tracing for behavior monitoring or profiling.

	SuperPI	ls	pwd	kill
MaLT [13]	192	595	134	n/a
Ninja [14]	1	n/a	n/a	n/a
OASIS w/o preprocessing	195	99	82	72
OASIS w/ preprocessing	183	77	62	54

TABLE II. Times of slowdown on test cases

B. Case Study II: Kernel Analysis With Fuzzing and EFI

Existing kernel fuzzing tools [35], [36], [37], [20] rely on kernel code instrumentation to collect runtime intelligence. For instance, Google Syzkaller [20] uses the kernel address sanitizer (KASAN) [38] to validate and report memory accesses and uses `ftrace` to log kernel function calls. This approach has two limitations. Firstly, it is not adaptive enough to meet different analysis demands because the concerned code regions 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 general approach (as shown in Figure 10) is to export the Syz-executor to the onsite environment where the EFI analyzer makes on-demand analysis and gleans runtime data inaccessible to the instrumentation code.

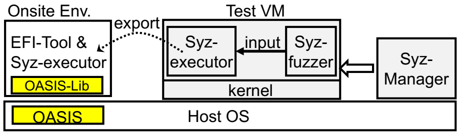


Fig. 10. An EFI analyzer works in tandem with Syzkaller (the gray boxes).

1) Dynamic Postmortem Analysis: In our fuzzing test, one reported reproducible crash is caused by a page fault when KASAN validates the address 0xffff880869a0affc which is accessed by the kernel function `ata_bmdma_fill_sg()`.[3] The instruction under validation is to update the `flaglen` member in `ata_bmdma_prd[pi-1]`. To understand the build-up of the page fault, we develop an EFI analyzer to collect runtime data from the reproduced execution with the strategy formed by a static analysis.

To choose the EFI instrumentation junctures, we make a backward slicing upon the instruction under validation. We then determine what to introspect at each juncture by checking the source code correlating to the sliced instructions.

[3] The function is in the default ACSI device driver in the kernel.

The objects to collect mainly include the input parameter to `ata_bmdma_fill_sg()`, the variable `pi`, objects determining the control flow, and memory data used to calculate the address under checking. 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 using the JMP-probe until the page fault occurs. For blocks containing the sliced instructions, it 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. The data collected from the last three sliced instructions shows that 0xffffffff is used to derive 0xffff880869a0affc as the address for `ata_bmdma_prd[pi-1].flaglen`. The address matches with the crash report that its validation 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 (provided by the fuzzer) is well-formed.

2) Exploration of Untrusted Driver: Our second analyzer runs with Syzkaller to uncover hidden behaviors of an uninstrumented 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 malicious driver with stealthy behaviors built on a rootkit in Github [4]. 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. Since the driver is randomly loaded due to ASLR used in kernel bootup, the analyzer locates the handler 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. For each exported Syz-executor, the analyzer installs an INT-3 probe at the entrance of the driver's `ioctl` handler. When the probe is triggered, the analyzer removes it and installs one 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. It is reported that the handler's hidden path executes 65 basic blocks and 7 of calls to kernel functions including `prepare_cred()` and `commit_creds()`.

Summary. The two experiments showcase EFI's agility in controlling and introspecting a kernel thread. EFI analyzers are well-suited for on-demand and fine-grained analysis as the analyst can flexibly determine the strategy to deploy the two types of probes. It is not difficult to develop them as they are user-space programs and can benefit from existing libraries.

IX. Experiments

A. OASIS Performance

To understand the hardware characteristics, we measure the CPU time taken by relevant hardware events and instructions in our platform excluding software processing. The results are reported in Table III which shows that VM-exit incurs much higher overhead than other events. Table IV reports the time costs of five popular system calls issued in the onsite environment and in the host. In average, the overhead is around 2.21 μs per system call, largely due to one round VM exit/entry and one debug exception.

INT3	vmfunc	iret	VM Exit + Enter
340	147	367	916

TABLE III. CPU CYCLES FOR RELEVANT EVENTS AND INSTRUCTIONS

	open	read (4KB)	write (4KB)	brk	getpid
From host	1.41	0.79	1.14	0.84	0.39
From onsite env.	3.55	2.97	3.50	3.04	2.56
Overhead	2.14	2.18	2.36	2.20	2.17

TABLE IV. OVERHEAD IN ANALYZER'S POPULAR SYSCALLS (IN μs)

The EFI overhead is characterized by the average time for a round-trip control flow transition between the target and the analyzer. Since the INT3-probe and the JMP-probe work differently, we evaluate them separately. Following the method in SPIDER [12], we first measure the time difference between the target execution without EFI and its execution with a null EFI tracer (i.e., no payload function), and then divide it by the number of round-trips made during tracing. The result is in Table V including the data reported in the literature for the hypervisor-based approach [12] and the SMM-based approach [13]. The dominant overhead of using the JMP-probe includes `vmfunc`, 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 as fast as those in SPIDER [12]. The overhead SPIDER is attributed to the hassle of single-stepping the restored instruction while the overhead in MalT [13] is entirely due to the hardware.

Transition	JMP-probe	INT3 Trap to Hyp [12]	Trap to SMM [13]
Cost	836	3,217	28,128

TABLE V. ROUND-TRIP TRANSITION OVERHEAD USING JMP-PROBE (IN CPU CYCLES).

The transition overhead for the INT3-probe comprises the INT#3 exception, `vmfunc`, and the make-up execution of the affected instruction. The exception overhead varies with the probe's CPU privilege, while the make-up overhead varies with

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

the types of the overwritten instructions since different methods are used for make-up execution. Table VI 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 VI. ROUND-TRIP TRANSITION OVERHEADS USING INT3-PROBE (IN CPU CYCLES).

B. Benchmark Testing

We evaluate the impact of EFI upon the guest kernel, by running the LMbench tools [39] 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 two experiments with different 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 analysis. Although the analyzer does not consume hardware resources of the guest, the slowdown of the captured thread may affect other threads due to synchronization or resource sharing.

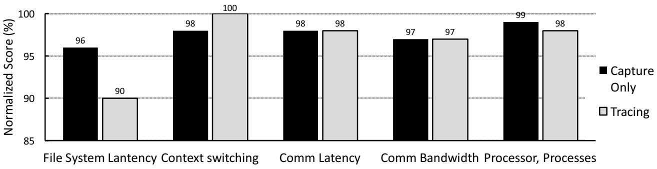


Fig. 11. Normalized LMbench results (in % of the native benchmark results).

The normalized results on all system aspects of LMbench in two experiments are shown in Figure 11. Due to paper length limit, we omit the detailed report of each category. Except File System Latency showing significant performance slowdown (4% and 10% in two experiments), other four benchmarks only report up to 3% drop. A deeper investigation on the File System Latency data reveals that it is largely due to page fault handling whose performance drops 12.7% and 28.6% 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.

C. 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 [9]. The third is to obstruct analysis by code and data obfuscation, usually via packing/unpacking [40]. A sophisticated malware may use a combination of them.

1) System Environment Detection: Existing techniques include anti-emulation, anti-sandbox, anti-VM 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 waning 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 [41], an off-the-shelf program implementing popular anti-debugging techniques used by Linux malware [42], [43]. We analyze it using both EFI tracing and EFI breakpoints. The results are in Table VII.

Technique	Brief description	Result
`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 VII. ANTI-DEBUGGING TECHNIQUES.

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. The `rdtsc` attack is essentially a side-channel attack. We 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 [13], an ultimate solution may require fine-grained instruction level semantics.

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 a self-checksum program in Github[5] which calculates a hash digest of its executable and read-only data memory. We run it under GDB with and without a breakpoint at its main function. Both checksums are different from the original one calculated in the native execution. In contrast, it still reports the original result when undergoing EFI analysis using the INT3-probes.

Kirsch et. al. [9] propose thirteen methods to detect DBI tools including Pin [22], DynamoRIO [44] and Valgrind [23]. We test OASIS against their methods integrated in *jitmenot* [45]. Our EFI tracing and breakpoint tools pass all the detection methods listed in Table VIII.

The methods are grouped to three categories [9]. The first seven techniques check code cache/instrumentation artifacts. They cannot break OASIS transparency, because OASIS does

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

Technique	Brief description	Result
`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 VIII. REPRESENTATIVE ANTI-INSTRUMENT TECHNIQUES.

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. Although EFI tools indeed incur longer delays, they are not detected by the attacks because they are much shorter than JIT compilation. The last four techniques are in the category of runtime environment artifacts. Although OASIS relocates the target's IDT, GDT and TSS, they are transparent to the target.

3) Packer: According to Cozzi et. al. [40], the most popular packer used by Linux malware is UPX [46] which is a Type-I packer [47]. Unpacking is essentially malware's dynamic modification in its own virtual address space. It has no adverse impact on OASIS. The paging hierarchies in the onsite-environment are designed to be consistent with the one in the guest. EPT updates in the guest are also cloned to the onsite-environment. The unpacking process may read or write the virtual page with probes or change its mapping to another physical page. We have explained how OASIS protects probes in these two scenarios in §VI-A, namely, by leveraging d-TLB and i-TLB to cope with read/write accesses and by monitoring GPA update to cope with mapping changes.

No. of basic blks	No. of syscalls	No. of #PFs	No. of self-writes
311,994	38	51	388

TABLE IX. EFI-TRACING FOR UPX PACKED UNAME.

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 IX. 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. 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.

X. Discussions and Conclusion

Paralleled Analysis. OASIS can be extended to launch a multicore onsite environment for two or more analyzer threads. One potential 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. Another 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.

Target Control Without Probes. Hardware facilities such as PMU can be applied in OASIS to control the target execution. 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 tames the target execution in order to acquire fine-grained software semantics, while the trapping-based EFI is more suitable for event-centric analysis. The two EFI approaches only differ in target control, i.e., how the interleaving juncture is chosen and triggered.

ARM Platform. OASIS can be exported to ARM platforms with modest changes since the guest table walking also requires GPA-to-HPA translation under ARM virtualization. Hence, the analyzer/target hierarchy and the target/lib hierarchy can be constructed on an ARM processor. The key difference is that the ARM architecture does not have a Ring 3 instruction equivalent to Intel's `vmfunc` which allows the analyzer to switch the underlying GPA-to-HPA translation table. Hence, interleaving the target and the analyzer instruction flows mandates privilege level switches. Nevertheless, due to different virtualization strategies, the switch in ARM is much faster (about 300 CPU cycles in our experiments) than VM-exit and VM-entry in x86-64 platforms.

Conclusion. To summarize, we introduce the notion of EFI and the infrastructure OASIS to support it. OASIS based EFI combines the advantages of hardware based event-trapping (i.e., strong security and non-intrusiveness) and code instrumentation (i.e., native-access) without their disadvantages. The foundation of OASIS is to securely fuse the target's paging hierarchy with the analyzer's by using virtualization techniques. Both the EFI analyzer and the target run in the onsite environment launched by OASIS. The analyzer uses software probes to choose the junctures to instrument the target execution flow. Our case studies demonstrate that it is not difficult to master EFI to develop nimble tools analyzing a target thread running across user and kernel modes. We have also rigorously conducted experiments to evaluate OASIS security, transparency and performance. The source of code of OASIS and EFI tools is available at *https://github.com/OnsiteAnalysis/OASIS*.

Acknowledgement

This article is partially supported by the National Research Foundation (NRF) Singapore and National Satellite of Excellence in Trustworthy Software Systems (NSoE-TSS) award number NSOE-TSS2019-04.

Appendix A
Background on Address Translation and Virtualization

A. Address Translation without MMU Virtualization

Software on an x64 platform typically uses 48-bit virtual addresses (VAs) which cover 256 Terabytes in total [48]. 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.

As shown in Figure 12, a 48-bit virtual address is divided into four 9-bit indexes and one 12-bit in-page offset. The four indexes are used to select the entries in PML4, PDPT, PD and PT, respectively. Since the design of OASIS uses the first 9-bit index frequently, we name it as the *PML4 prefix* of a virtual address for ease of presentation.

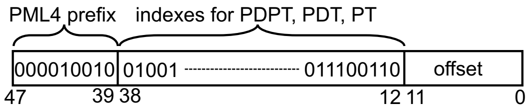


Fig. 12. Illustration of a 48-bit virtual address on a 64-bit platform. Its PML4 prefix is “00001010”.

To translate a virtual address X, the MMU uses `CR3` to locate the physical location of the PML4 table in use. X’s PML4 prefix 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.

B. 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 the next level paging structure page with its GPA, the MMU traverses the Extended Page Tables (EPT) to locate the corresponding HPA. Hence, to translate a virtual address, the MMU must consult the EPTs for four times in order to locate and read 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 13.

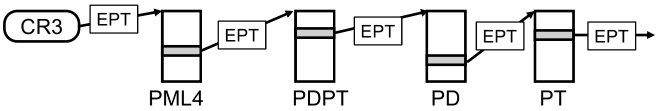


Fig. 13. The EPTs are consulted for four times when the MMU visits four levels of paging structures and one more time for locating the final physical page.

Hence, the EPTs play a pivotal role of 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.

Appendix B
Case Study Implementation Report

To support EFI analyzer implementation, we develop OASIS APIs listed in Table X. 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 XI reports their source code sizes.

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 X. 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 XI. Source code size of analyzers.

In the following, we report more implementation details of the postmortem analyzer. Table XII 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 defined at line 2615 of kernel source file libata-sff.c.The postmortem analyzer mainly

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 XII. VIRTUAL MEMORY LAYOUT OF THE POSTMORTEM ANALYZER.

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 control 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.

```
 1 void ana_int3_handler (void)
 2 {
 3   ...
 4   // if it is the target function invoked
 5   if (target_ctx->rip == BP1) {
 6
 7     //remove the INT3-probe
 8     onsite_rm_int3_probe(BP1);
 9
10     // target introspection
11     qc = target_ctx->rdi;
12     ap = qc->ap;
13     prd = ap->bmdma_prd;
14     ...
15
16     //traverse and dump kernel object list
17     sg = qc->sg;
18     for (i = 0; i < qc->n_elem; i ++)
19     {
20       sg_len = sg->length;
21       sg_dma_addr = sg->dma_address;
22       fprintf (fp, ‘‘sg at: %p,,... ”, ...);
23       sg ++;
24     }
25
26     /* find next block exit */
27     blkExit = onsite_find_n_exit(BP1);
28
29     //assign probAddr: blkExit or slice
30     ...
31
32     // install JMP-probe
33     onsite_install_trace_probe(probAddr);
34
35   }
36   onsite_t_run (); //resume target;
37   return;
38 }
```

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 the 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;
}
```

References

[1] 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.

[2] 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.

[3] B. Chamith, B. J. Svensson, L. Dalessandro, and R. R. Newton, "Instruction punning: Lightweight instrumentation for x86-64," in *Proceedings of ACM Conference on Programming Language Design and Implementation (PLDI)*, 2017.

[4] 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.

[5] D. B. I. F. I. know you're there spying on me, "Francisco falcó and nahuel riva," in *REcon*, 2012.

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

[15] "ATRA: Address translation redirection attack against hardware-based external monitors," in *Proceedings of the 2014 ACM Conference on Computer and Communications Security (CCS)*, 2014.

[16] "Vigilare: toward snoop-based kernel integrity monitor," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS)*, 2012.

[17] 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, 01 2014.

[18] 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.

[19] 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.

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

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

[22] 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.

[23] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *Proceedings of the 2007 ACM Sigplan Conference on Programming Language Design and Implementation (PLDI)*, 2007.

[24] A. R. Bernat and B. P. Miller, "Anywhere, any-time binary instrumentation," in *Proceedings of the 10th ACM SIGPLAN–SIGSOFT Workshop on Program analysis for Software Tools*, 2011, pp. 9–16.

[25] 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.

[26] M. Wenzl, G. Merzdovnik, J. Ullrich, and E. Weippl, "From hack to elaborate technique – a survey on binary rewriting," *ACM Computing Surveys (CSUR)*, vol. 52, 2019.

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

[28] 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*. ACM, 2011, pp. 363–374.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

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

[35] 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 ACM Communications and Computer Security (CCS)*, 2017.

[36] 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 *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2019.

[37] W. Wu, Y. Chen, J. Xu, X. Xing, X. Gong, and W. Zou, "FUZE: Towards facilitating exploit generation for kernel user-after-free vulnerabilities," in *Proceedings of the 27th USENIX Security Symposium*, 2018.

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

[39] LMbench - tools for performance analysis. [Online]. Available: http://lmbench.sourceforge.net/

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

[41] Pangu. [Online]. Available: https://github.com/jvoisin/pangu

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

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

[44] D. Bruening, T. Garnett, and S. Amarasinghe, "An infrastructure for adaptive dynamic optimization," in *Proceeding of the IEEE International Symposium on Code Generation and Optimization (CGO)*, 2003.

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

[46] The ultimate packer for eXecutables. [Online]. Available: https://upx.github.io/

[47] 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 *Proceedings of the 2015 IEEE Symposium on Security and Privacy (S&P)*, 2015.

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