

METHOD AND SYSTEM FOR DYNAMICALLY DETECTING, ANALYZING, MONITORING, INVESTIGATING AND/OR EXECUTING A LIVE TARGET PROGRAM

FIELD OF THE INVENTION

5

[001] The present invention relates generally to virtual address space introspection, and more specifically to dynamically detecting, analyzing, monitoring, investigating and/or executing a live target program through its live address space.

10 BACKGROUND OF THE INVENTION

[002] A virtual address space (VAS) is a set of ranges of virtual addresses provided to a program by an operating system (OS). The VAS of a running program contains its code and static/dynamic data objects, as well as their virtual addresses. Because these memory-resident runtime artifacts provide
15 the basis to understand software logic and to acquire high-level intelligences, they are instrumental to live analysis and/or investigation on running programs, such as dynamic malware analysis, live forensics, real-time monitoring and virtual machine (VM) introspection.

[003] However, it is unfortunately not easy for a detecting, analyzing, monitoring, investigating and/or
20 executing tool (hereinafter referred to as analyzer) to make full-scale on-demand accesses to a live address space of a target program (hereinafter referred to as target). One of the issues is the address space barrier between two programs, which makes it difficult to access one program's virtual address space by another program because their virtual addresses are mapped differently by the OS. Because the target could be an untrusted or even malicious program, another issue is that dropping the analyzer
25 directly into the target's live space exposes the former to potential attacks.

[004] For the convenience of presentation, virtual address (VA) segment belonging to the analyzer's code and data is referred to as the local address space; and VA segment belonging to the target is referred to as the target address space. To resolve the above issues, existing methods are known to
30 either 'take snapshots' of the target's physical memory regions, or to re-map them into the analyzer's local address space. Although both approaches are secure, they are cumbersome and costly. Further, due to pre-processing overhead (see paragraph [046] below), neither of these two methods is ideal for on demand accesses. Another disadvantage of these two methods is that it cannot be guaranteed that the mappings used by the analyzer's accesses are consistent with that of the target, which undermines
35 the validity of virtual address related intelligence.

[005] To resolve the abovementioned issues, Siqi Zhao et al. (<https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/zhao>) proposed an Immersive Execution Environment (ImEE), which is a special computing environment with a twisted
40 address mapping setting. The ImEE's CR3 is synchronized with the target VM's active CR3 so that its

Memory Management Unit (MMU) directly uses the target's VA to guest physical address (GPA) mappings. Its GPA to host physical address (HPA) mappings are split into two: the GPAs for the intended introspection are translated with the same mappings as in the target VM; the GPAs for the local usage are mapped to the local physical pages via separated GPA-to-HPA mappings. It allows the

5 so-called ImEE agent, which is specially crafted with memory loading instructions only, to execute in the same address space with the target. The agent in the ImEE environment reads directly from the target's address space with guaranteed consistency and at native speed. The memory-read results are saved into CPU registers for a separated thread to fetch.

10 [006] However, as ImEE imposes strict restrictions on the agent, such as no system calls and no local data access, it is incapable of running a regular application, e.g. analyzing the information and taking necessary actions against the target etc. Therefore, the ImEE system can only partially solve the aforementioned problems due to these restrictions.

15 [007] Therefore, there is a need to provide a method and system supporting a full-fledged analyzer program to execute in a target's virtual address space, so that: the analyzer's introspection to the target is guaranteed with mapping consistency and is at native speed; and the analyzer appears as a normal user space application to the host OS and receives its services, e.g. being able to run a regular application, to analyze and take necessary actions against the target.

20 SUMMARY OF INVENTION

[008] According to an aspect of the present invention, there is a computer-implemented method for detecting, analyzing, monitoring, investigating and/or executing a target program, the method

25 comprising: obtaining a tool paging structure of a tool, wherein the tool paging structure is created by a host operating system; obtaining a target paging structure of the target program, wherein the target paging structure is created by a target virtual machine; and generating a hybrid paging structure by grafting at least one part of the tool paging structure into at least one part of the target paging structure; wherein the hybrid paging structure is used for virtual address translation when detecting, analyzing,

30 monitoring, investigating and/or executing the target program using the tool.

[009] According to another aspect of the present invention, there is a computer program comprising instructions which, when the program is executed by a computer, cause the computer to carry out a computer-implemented method for detecting, analyzing, monitoring, investigating and/or executing a

35 target program, the method comprising: obtaining a tool paging structure of a tool, wherein the tool paging structure is created by a host operating system; obtaining a target paging structure of the target program, wherein the target paging structure is created by a target virtual machine; and generating a hybrid paging structure by grafting at least one part of the tool paging structure into at least one part of the target paging structure; wherein the hybrid paging structure is used for virtual address translation

40 when detecting, analyzing, monitoring, investigating and/or executing the target program using the tool.

[010] According to another aspect of the present invention, there is a computer-readable medium comprising instructions which, when executed by a computer, cause the computer to carry out a computer-implemented method for detecting, analyzing, monitoring, investigating and/or executing a target program, the method comprising: obtaining a tool paging structure of a tool, wherein the tool
5 paging structure is created by a host operating system; obtaining a target paging structure of the target program, wherein the target paging structure is created by a target virtual machine; and generating a hybrid paging structure by grafting at least one part of the tool paging structure into at least one part of the target paging structure; wherein the hybrid paging structure is used for virtual address translation when detecting, analyzing, monitoring, investigating and/or executing the target program using the tool.

10 [011] According to another aspect of the present invention, there is a system for detecting, analyzing, monitoring, investigating and/or executing a target program, the system comprising: a host operating system, a target virtual machine and a tool; wherein a tool paging structure is provided to the tool by the host operating system; wherein a target paging structure is provided to the target program by the
15 target virtual machine; wherein a hybrid paging structure is generated by grafting at least one part of the tool paging structure into at least one part of the target paging structure; and wherein the hybrid paging structure is used for virtual address translation when detecting, analyzing, monitoring, investigating and/or executing the target program using the tool.

20 [012] Preferably, the tool appears as a normal application to the host operating system when receiving service from the host operating system and accesses a virtual address space of the target program with the same paging structure as the target program when detecting, analyzing, monitoring, investigating and/or executing the target program.

25 [013] Preferably, the target program is a malware or a benign software (including the OS).

[014] Preferably, the system comprises transactional memory.

30 [015] Preferably, the tool comprises at least a static analysis tool and/or at least a dynamic analysis tool.

[016] Preferably, the tool detects, analyzes, monitors, investigates and/or executes the target program at runtime.

35 [017] Preferably, the tool is a position-independent executable.

[018] Preferably, when the hybrid paging structure is used for virtual address translation, different access permissions (e.g. read, write and execute permissions) are selected to be set on pages storing the target program's code and data according to 1) level of security required at the time of translating,
40 2) trust level of the pages storing the target program's code and data, and 3) the tool's needs.

[019] Preferably, when the hybrid paging structure is used for virtual address translation, execute permission is set on pages storing the tool's code.

[020] Preferably, when the hybrid paging structure is used for virtual address translation, at least one of read and write permissions is selected to be set on pages storing the tool's code according to 1) level of security required at the time of translating and 2) trust level of the pages storing the tool's code.

BRIEF DESCRIPTION OF THE DRAWINGS

[021] Figure 1 depicts a 48-bit virtual address on a 64-bit platform.

[022] Figure 2 depicts a four-level paging structure and their consultation to Extended Page Tables (EPT).

[023] Figure 3 depicts the architectural view of the present invention according to one of the preferred embodiments of the present invention.

[024] Figure 4 depicts a hybrid paging structure and the source paging structures thereof according to one of the preferred embodiments of the present invention.

[025] Figure 5 depicts the mapping of a hybrid paging structure according to one of the preferred embodiments of the present invention.

[026] Figure 6 depicts an overview of the paging structure in onsite mode according to one of the preferred embodiments of the present invention.

[027] Figure 7 depicts the comparison between normal mode switch and the mode switch according to one of the preferred embodiments of the present invention.

[028] Figure 8 depicts the workflow of mode switches according to one of the preferred embodiments of the present invention.

[029] Figure 9 depicts a potential Page Map Level 4 (PML4) manipulation attack.

[030] Figure 10 depicts the visualization of pointers discovered on a stack page according to one of the application examples of the present invention.

[031] Figure 11 depicts the evolution of kernel stack from receiving *syscall* to completion of *sysret* according to one of the application examples of the present invention.

DETAILED DESCRIPTION OF THE INVENTION

[032] While the concepts of the present disclosure are susceptible to various modifications and alternative forms, specific embodiments thereof have been shown by way of example in the drawings and will herein be described in detail. It should be understood, however, that there is no intent to limit the concepts of the present disclosure to the particular forms disclosed or terms used, but on the contrary, the intention is to cover all modifications, equivalents, and alternatives consistent with the present disclosure and the appended claims.

[033] In the following description, for the purposes of explanation, numerous specific details are set forth in order to provide a thorough understanding of the present invention. It will be apparent, however, that the present invention may be practiced without these specific details. Well-known methods, procedures and components have not been described in detail so as not to unnecessarily obscure aspects of the implementations.

[034] In the drawings, specific arrangements or orderings of schematic elements, such as those representing devices, modules, instruction blocks and data elements, may be shown for ease of description. However, it should be understood by those skilled in the art that the specific ordering or arrangement of the schematic elements in the drawings is not meant to imply that a particular order or sequence of processing, or separation of processes, is required. Further, the inclusion of a schematic element in a drawing is not meant to imply that such element is required in all embodiments or that the features represented by such element may not be included in or combined with other elements in some embodiments.

[035] Further, in the drawings, where connecting elements, such as solid or dashed lines or arrows, are used to illustrate a connection, relationship or association between or among two or more other schematic elements, the absence of any such connecting elements is not meant to imply that no connection, relationship or association can exist. In other words, some connections, relationships or associations between elements may not be shown in the drawings so as not to obscure the disclosure. In addition, for ease of illustration, a single connecting element may be used to represent multiple connections, relationships or associations between elements. For example, where a connecting element represents a communication of signals, data or instructions, it should be understood by those skilled in the art that such element may represent one or multiple signal paths (e.g., a bus), as may be needed, to affect the communication.

[036] Several features are described hereafter that can each be used independently of one another or with any combination of other features. However, any individual feature may not address any of the problems discussed above or might only address one of the problems discussed above. Some of the problems discussed above might not be fully addressed by any of the features described herein.

Although headings are provided, information related to a particular heading, but not found in the section having that heading, may also be found elsewhere in the specification.

[037] It will also be appreciated that, although the terms first, second, etc. may be used herein to describe various elements, these elements should not be limited by these terms. These terms are only used to distinguish one element from another. For example, a first item could be termed a second item, and, similarly, a second item could be termed a first item and so on.

[038] The terms “comprise”, “contain” and “include” are generally interpreted as being open-ended, which means other alternatives might be included, no matter which form they are in. For example, “a combination comprising components A and B” would include a combination of components A, B and C. It will also be appreciated that the singular forms “a”, “an” and “the” are intended to include the plural forms as well, unless the context clearly indicates otherwise. It will further be understood that the term “and/or” as used herein refers to and encompasses any and all possible combinations of one or multiple of the associated listed items.

[039] The definitions of some terms are provided below which apply throughout the present invention. Unless defined otherwise, all technical and scientific terms used herein have the same meanings as commonly understood by those skilled in the art to which the present invention belongs.

[040] “Onsite mode” is a virtual machine extension non-root mode in which the analyzer program runs in the same live virtual address space as the target program. The onsite mode may be used for introspecting the target’s address space and other types of dynamic analysis.

[041] “Host mode” is a virtual machine extension root mode in which the analyzer program appears as a regular process to the host OS. The host mode may be used for the host OS to provide kernel services under the same thread context.

[042] “Trampoline” is a host kernel module which handles the transition between onsite mode and host mode. For example, based on an exception, a trampoline can switch the working mode, and update parameters etc.

[043] “Manager” is a module for controlling hardware/software resources for onsite mode. For example, a manager can allocate and configure the needed hardware and software infrastructure. It can also update the information if there are any changes.

[044] “Cross-domain mapping consistency” is the consistency between the translation of any VA in the target address space in onsite mode and the translation of the same VA in the target VM. In other words, with this consistency, the MMU in onsite mode translates the VA in the target address space to

the same physical address as the MMU does in the target VM. Cross-domain mapping consistency implies the correctness of accessing the target virtual address space.

[045] “Cross-mode mapping consistency” is the consistency between translation of any VA in the local address space in onsite mode and the translation of the same VA in host mode. Cross-mode mapping consistency implies that the host kernel’s execution in host mode shares the same memory view as the analyzer in onsite mode. It ensures that the host kernel provides the correct service to the analyzer under the proper context.

[046] “Grafting” is a process of replacing an undesired part of information with a desired part of information. For example, if a sequence A includes information A1, A2 and A3, and a structure B includes information B1, B2 and B3, sequence A may be grafted by replacing any piece of information from sequence A with any piece of information from structure B, e.g. (A1, A2 and B1), (A1, A2 and B3), or (B2, A2, B3) etc.

[047] “Overhead” is the additional time spent for taking one or more actions, compared to the case that no such action/s is/are taken or where such action/s is/are taken under conditions which require less time. For example, the overhead of action A means the additional time required for taking action A compared to the case that action A is not taken. As another example where action B is taken in two Modes X and Y, the overhead of action B in Mode X means the additional time spent for taking action B in Mode X compared to the lesser time spent for taking action B in Mode Y.

[048] It is provided an Onsite AnalySis InfraStructure (OASIS) in the present invention that allows a full-fledged analyzer application to securely and transparently execute in the live address space of a target. Generally, OASIS changes the EPT mapping to redirect the MMU to a different Page-Directory-Pointer-Table (PDPT) page without modifying the PML4 entry. In other words, OASIS provides a hybrid paging structure, part of which is obtained from the target while the other part is obtained from the analyzer. OASIS makes no modification on the analyzer binary, the target binary, or the target environment including the libraries and the OS. With the hybrid paging structure, the analyzer smoothly receives OS services as normal and accesses the target’s space with guaranteed mapping consistency and at native speed. The OASIS is further described in detail as below.

[049] In some of the preferred embodiments of the present invention, OASIS runs on a platform with CPU and MMU virtualization extensions. In some of the preferred embodiments of the present invention, the platform is an x86 64-bit platform. The platform may be a commodity platform, an experimental platform or any other types of platform. The platform may contain one or more cores. The platform is managed by a host OS. In some of the preferred embodiments of the present invention, the host OS may be a bare-metal hypervisor, or a hypervisor runs within a conventional operating system environment. Some examples of the bare-metal hypervisors are Xen, VMware ESX, Microsoft

Hyper-V, KVM and Virtual PC. Some examples of the conventional operating systems are Microsoft Windows®, Linux®, Unix®, Solaris® X86 and macOS®.

[050] A VM hosting the target software is denoted as the target VM, and its OS is denoted as the target OS. Generally, the hardware, the firmware and the host OS are trusted, but the target can tamper with the code and data of the target OS and hence the target OS may not be trusted. In some embodiments of the present invention, the target VM may be the same as those VM hosted in a cloud platform or personal computers. OASIS does not impose any restriction or require changes on the VM's hardware or software. The platform administrator manages peripheral device assignment to VMs and handles malware containment.

[051] The analyzer may be a Position-independent executable (PIE), which does not use absolute addresses in its code or static data. The target may be any application or software, such as a malware or a normal software to be detected, analyzed, monitored, investigated and/or executed. The malware may be virus, spy ware, worm or any other types of malicious software. The software can also be a benign application or software, such as a computer game, a website browser or an OS.

[052] Any computer program may comprise instructions which, when the program is executed by a computer may cause the computer to carry out any OASIS-based detection, analysis, monitoring, investigation and/or execution of a target program.

[053] Any computer-readable medium may comprise instructions which, when executed by a computer may cause the computer to carry out any OASIS-based detection, analysis, monitoring, investigation and/or execution of a target program.

[054] For ease of describing the present invention, according to some of the preferred embodiments of the present invention, the OASIS based on an x86 64-bit Platform managed by Linux KVM occupying CPUs in the virtual machine extension (VMX) root mode is described as below; it will be appreciated that the OASS must not be limited to such platform and OS. In these embodiments, the host OS also plays the role of a hypervisor. Further, a target is launched in a VM whose CPUs are in the VMX non-root mode.

[055] Generally, there are two typical ways of translating address in x86_64 platforms, i.e. translating address with or without MMU virtualization, which are summarized as below.

[056] **Address Translation without MMU Virtualization.** Typically, 48-bit VAs which cover 256 terabytes in total are used for software on an x86 64-bit platform. While the higher half of the address space is occupied by the kernel, the other half is for user space. Usually, the paging structure on 64-bit platforms consists of four levels of paging structures. The top level is the PML4 table whose base address is stored in the CR3 register. The other three levels are the PDPT, the Page-Directory (PD),

and the Page-Table (PT). Each page of these paging structures contains 512 entries indexed by 9 bits in a VA. An entry of a PML4, PDPT 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 1, 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. For ease of reference, the first 9-bit index is named as the *PML4 prefix* of a virtual address. 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.

[057] **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 structure using the four 9-bit indexes, respectively. The main difference is that all addresses in CR3 and the four-level paging structures are GPAs, instead of physical addresses. To access a GPA, the MMU traverses the EPT to locate the corresponding physical address. Hence, to translate a virtual address, the MMU must consult the EPT for four times in order to walk the PML4, PDPT, PD, and PT pages, and for the fifth time to get the physical page frame number. The involvement of the EPT in address translation is illustrated in Figure 2.

[058] Figure 3 depicts a high-level architectural view of OASIS 300 according to one of the preferred embodiments of the present invention. In this embodiment, the system comprises at least an analyzer 305, at least a target VM 301 and at least a host OS 307. The host OS 307 comprises at least a trampoline 308 and at least an OASIS manager 309. The target VM 301 comprises at least a target 302 and a target OS 303. The analyzer thread lives in two modes: onsite mode 304 and host mode 306. The onsite mode 304 can be used for detecting, analyzing, monitoring, investigating and/or executing the target in the target's address space. The host mode 306 can be used for the host OS 307 to provide kernel services under the same thread context.

[059] The trampoline 308 handles switches between the onsite mode 304 and the host mode 306. The OASIS manager 309 controls hardware/software resources for onsite mode 304, e.g. the paging structures.

[060] Figure 4 depicts the analyzer 305 having its own local paging structure 401 created and managed by the host OS 307. Similarly, the target 302 has its own paging structure 403 in the target VM 301. According to an example of a paging structure as a tree of four levels with the root being pointed to by CR3, the hybrid paging structure 402 used in onsite mode 304 is provided by grafting one part from the analyzer's paging structure 401 into the target's paging structure 403. In other words, the EPT mapping is changed to redirect the MMU to a different PDPT page without modifying the PML4

entry. Hence, when the analyzer 305 runs in onsite mode 304, the MMU translates VAs belonging to the target space with the target's paging structure and translates VAs belonging to the local address space with the analyzer's paging structure. Both cross-domain and cross-mode mapping consistencies are achieved because of the correct paging structures in use. The analyzer 305 appears as a normal application to the host OS 307 when receiving service from the host OS 307 and accesses a virtual address space of the target 302 with the same paging structure as the target 302 when detecting, analyzing, monitoring, investigating and/or executing the target 302.

[061] Figure 5 depicts an example of the EPT redirection technique for grafting, which does not modify either the analyzer's paging structure 401 or the target's paging structure 403. The EPT is consulted by the MMU when traversing the paging structures. In this example, the PDPT page Y of the analyzer's paging structure 401 is grafted into the target's paging structure 403 at the 0x1F0-th entry of the target's PML4 table. The content of this entry is 0x1C00, which is the GPA of one of the PDPT pages in the target's paging structure 403. In the target VM 301, the EPT maps 0x1C00 to the physical address of Page X, which means Page X is what the PDPT page pointed to by the 0x1F0-th PML4 entry. If the same PML4 page is also to be used in onsite mode, the EPT is to be configured so that 0x1C00 is mapped to the physical address of Page Y. When translating any VA with its PML4 prefix being 0x1F0 in onsite mode, the MMU uses Page Y as the PDPT page, instead of Page X. Hence, different EPT settings in two environments result in different address mappings, although the same PML4 page is used by the hardware. For ease of expression, GPA being used to graft the PDPT page is referred to as Grafting-Site GPA, which is stored in one PML4 entry. For example, 0x1C00 is the Grafting-Site GPA in Figure 5. The paging structure grafting uses the EPT in onsite mode to redirect the Grafting-Site GPA in the target paging structure to a local paging structure page. Since no modification is made upon the target's paging structures, the grafting is fully oblivious to the target VM.

[062] In some of the preferred embodiments of the present invention, in host mode, the analyzer's local address space size is equal to or less than 512 GB and the host OS loads it to a 512-GB aligned base address. As a result, all local VAs have the same PML4 prefix and are translated via one PML4 entry. Hence, only one PDPT needs to be grafted to the target paging structure. One advantage is that the overhead of grafting is minimized. One other advantage is that the attack surface is minimized. For ease of presentation, taking λ_l as the 9-bit PML4 prefix of the base address, the analyzer in host mode occupies a 512 GB VA range $[2^{39}\lambda_l, 2^{39}\lambda_l + 2^{39} - 1]$. In some of the preferred embodiments of the present invention, this VA range is not used in onsite mode, and the analyzer is relocated into a different base address in onsite mode. For example, one 512 GB range used by the target kernel may be selected as the 'new home' for the analyzer. While a user space application on x86 64-bit platforms uses the lower half of the 48-bit space, relocating the analyzer into the kernel half provides an advantage of avoiding address range conflict with the target.

[063] It is appreciated that address relocation from host mode to onsite mode is compatible with paging structure grafting. Taking λ_l as the PML4 prefix of the analyzer's VAs in onsite mode, this means

that the PDPT page (e.g. Page Y in Figure 5) is linked to the λ_i -th entry of the PML4 table in host mode, and is grafted to the λ_t -th entry (e.g. $\lambda_t = 0x1F0$ in Figure 5) of the PML4 table in onsite mode.

[064] In some of the preferred embodiments of the present invention, the trampoline also adjusts the address related data in system call parameters and returns, which resolves the issues of: (1) although the physical locations of the analyzer's code and data are not changed across the modes (thanks to the grafting technique), the analyzer can only execute in onsite mode because address-related data generated at runtime could be invalid under the VA range in host mode; and (2) address relocation actually breaks cross-mode mapping consistency.

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

[066] Figure 6 depicts an overview of the paging structure in onsite mode. As the CPU in onsite mode runs in VMX non-root mode, VAs are firstly mapped into GPAs via the paging structure and then mapped to PAs via the EPT. The gray blocks in Figure 6 are the paging structures from the target thread, while the white blocks in Figure 6 are the analyzer's paging structures in host mode. The paging structure (i.e. the guest page tables) translates VAs to GPAs. No paging structure page is allocated in onsite mode. No paging structure pages from the target or the host is copied either. The entire paging structure is logically constructed using the paging structure pages in the target VM and in the host. Those pages are mapped to onsite mode by the EPT.

[067] The CR3 register is loaded with the base address of the target's PML4 table. Consequently, the MMU follows the same PML4 table as in the target to translate virtual addresses. On the PML4 table, the λ_t -th entry is the site chosen to graft the analyzer's PDPT page as shown in Figure 6. The PD and PT following the PDPT page are also annexed into the paging structure. As depicted in Figure 6, the analyzer's own code and data are referenced with VAs prefixed with λ_t . Hence, those VAs are translated by the MMU following the grafted paging structures pointed to by the λ_t -th PML4 entry.

[068] The EPT maps GPAs to physical pages. As explained above, all pages in the paging structure are either from the target VM or from the host. It implies that no physical page is allocated in onsite mode. All physical pages accessed by the hardware in onsite mode are either managed by the target OS or the host OS. The EPT maps them over to onsite mode, by mapping five sets of physical pages:

- (1) Pages storing the target's code and data;
- (2) Paging structure pages that map the target's code and data;

- (3) Pages storing the analyzer's data;
- (4) Pages storing the analyzer's code;
- (5) Paging structure pages that map the analyzer's code and data in host mode.

5 [069] Different access permissions can be selected to be set on each page according to 1) the level of security required at the time of translating; 2) the trust level of each page; and 3) the analyzer's needs. In one of the preferred embodiments of the present invention, access permissions include read permission, write permission and execute permission. The level of security may have different grades, and may be determined at the time of translating. For example, in a three-grade level of security system, 10 the grades can be indicated as low, middle and high; in a four-grade level of security system, the grades can be indicated as low, lower middle, upper middle and high. Similar to the level of security, the trust level of each page can also have different grades. It will be appreciated that grades can be defined in other ways or other terms, e.g. defined in digital number, or termed as trusted and untrusted. In some of the preferred embodiments of the present invention, the analyzer may only need to read the target, 15 or need to execute the target's code within onsite mode. The selection of access permission on a page can be made after considering the level of security, the trust level of this page, and the analyzer's needs. The "least privilege" principle should be applied to assign access permissions. For example, if the analyzer does not write to the target's page, that page's permission should not have the write permission.

20 [070] For the first page set, the EPT for these pages is cloned from the corresponding EPT used in the target VM. Since the target's paging structure is used directly in onsite mode, the same mappings on the EPT ensure that the VAs in the target space are mapped into the same physical pages as within the target VM. The read-only permission can be set on these pages, so that no target code can execute in onsite mode and the analyzer does not (inadvertently) modify the target's code and data.

25 [071] For the second page set, the corresponding EPT from the target VM with the same permission setting can be cloned, except the Grafting-Site GPA (e.g. GPA_{λ} in Figure 6). The EPT is configured so that GPA_{λ} is mapped to the physical address of the analyzer's PDPT page (e.g. Page Y in Figure 6).

30 [072] For the third page set, an identity map on the EPT can be created so that their physical addresses are the same as their GPAs. Because of paging structure grafting, the physical addresses of the analyzer's code and data in host mode become their GPAs. Hence, the identity map on the EPT maps the GPA back to the same PA. As a result, it is not needed to physically relocate the analyzer's code and data. The permissions on these pages are set in the same fashion as in host mode.

35 [073] For the fourth page set, random GPA-to-PA mappings in the EPT can be used to counter the mapping manipulation attack, which will be further discussed below. The analyzer code pages remain at their physical locations allocated in host mode. In one example, to realize random GPA-to-PA mappings, those page table entries (PTEs) pointing to executable pages may be modified so that they 40 point to GPA page numbers randomly chosen from $[2^{20} + 1, 2^{36} - 1]$. As the analyzer code does not

execute in host mode, modifications on PTEs for code page mappings do not disrupt host mode execution.

[074] For the fifth page set, the identity map on the EPT for all paging structure pages with the same permissions can be used. Hence, all these pages may be walked by the MMU in the same way as in host mode.

[075] With the EPT configured as described above, the MMU in onsite mode follows the expected path to walk the paging structure, which ensures both cross-domain and cross-mode mapping consistencies. At runtime, the host OS may update the analyzer's paging structures to alter its virtual address space due to memory allocations and deallocations. The updates will take the same effect in onsite mode, because those paging structures are also in used onsite mode. Nonetheless, when a new PD page or a new PT page is inserted in the structure, the OASIS manager needs to update the onsite mode EPT accordingly.

[076] In one of the preferred embodiments of the present invention, the host OS is configured to assign $[0, 2^{32} - 1]$ as the GPA range for the target VM. Hence, the first two sets of pages all have GPAs between 0 and 4G. For the third and fifth sets, the host OS randomly chooses the physical pages from the pool between 4G to the maximum physical address available in the platform. For the fourth page set, the GPAs are randomized in onsite mode, not overlapping with the target's GPAs. This configuration provides an advantage that the GPA ranges of the first two page sets do not overlap with the other three page sets.

[077] Figure 7 depicts the comparison between normal mode switch and the OASIS's mode switch. Those skilled in the art would understand that a normal thread runs in either user-mode or kernel-mode and the CPU privilege changes along with mode switches. Switches between host mode and onsite mode in OASIS involve both the CPU privilege change and VMX mode change (root vs. non-root). When the analyzer thread is in onsite mode 304, the CPU runs with Ring 3 privilege under the VMX non-root mode. In this mode, the analyzer not only accesses its own code and data but also the target's address space. The whole VAS in onsite mode 304 is the union of one continuous VA segment belonging to the analyzer's code and data, and other VA segments belonging to the target 302.

[078] The analyzer in onsite mode 304 switches to host mode 306 via VM Exit which is triggered by system calls, exceptions and interrupts. In host mode 306, the host OS 307 handles the events, and the CPU runs with Ring 0 privilege under the VMX root mode. In other words, the host kernel executes under the analyzer's thread context to serve a system call request or to handle a hardware event. Hence, switches between host and onsite modes involve not only CPU privileges, but also VMX modes. After it finishes, it resumes the analyzer execution in onsite mode via VM Entry. In one of the preferred embodiments of the present invention, trampoline and OASIS manager are introduced in the host OS

to handle the mode switches. The trampoline not only facilitates the mode switch but also ensures the transparency of onsite mode to the host kernel.

[079] Figure 8 depicts the workflow of mode switches according to one of the preferred embodiments of the present invention. Switches between host and onsite modes do not change the thread context, which means there is no switch of task, which jointly with cross-mode mapping consistency, ensure correct kernel services offered to the analyzer.

[080] It is appreciated that the main tasks for switching to host mode from onsite mode are 1) to set up the proper context for the kernel to process; and 2) to redirect the kernel execution flow to onsite mode once the service is done.

[081] Step 1 in Figure 8 depicts a VM Exit. Since onsite-to-host switch is essentially a VM Exit, the hardware behaves differently from a privilege switch caused by system calls, interrupts and exceptions. When a VM Exit is triggered, the hardware saves the analyzer's vCPU registers into the Virtual Machine Control Structure (VMCS), switches the CPU mode to the VMX root mode, and passes the control to the trampoline which is set as the default VM Exit handler for the analyzer in advance. The trampoline then adjusts the context as if the event were from the host.

[082] System calls and interrupts are easy to handle because only registers are involved. Their occurrences in onsite mode trigger a VM Exit because an EPT violation is raised when the hardware attempts to load the kernel handler. Note that the EPT violation exception does not affect those registers storing the context for system calls and interrupts. Hence, the trampoline makes no context adaption except loading the physical CPU's registers according to the VMCS.

[083] It is slightly more complex to adapt the context for exceptions. When handling a user-space exception in a normal setting, the hardware uses the kernel stack and some registers to provide the necessary data for the kernel to resolve the issue. Since no kernel stack is in onsite mode, the trampoline follows the hardware convention to fill in the kernel stack and load relevant registers.

[084] Once the event context is ready, the trampoline invokes the corresponding OS handler in a 'returning-boomerang fashion', which is to regain the control after the service is completed. It first sets a hardware breakpoint to intercept the kernel's return to user-mode of the analyzer thread, and then *jumps* to the entry of the kernel handler, as Step 2 in Figure 8. Note that function calls cannot be used to invoke the kernel service because the kernel handlers are not supposed to be called by software.

[085] With a properly prepared event context, the host OS can smoothly service the event in question. Unless the analyzer thread is aborted, the kernel eventually attempts to return to the user mode of the thread. Note that onsite mode is transparent to the kernel. Hence, the breakpoint hijacks the control flow right after the privilege switch so that the trampoline regains the CPU and returns to onsite mode.

[086] Signal handling requires a special treatment since the kernel needs to call the user-space signal handler if any. To maintain the semantics of signals, a dummy user-space signal handler 801 is registered to the host OS if the analyzer has any non-default signal handler. When invoked by the OS, the dummy handler 801 saves the parameters (if any) and sets a flag to indicate its invocation. The analyzer's non-default handler is then invoked after onsite mode execution resumes.

[087] Step 3 and step 4 in Figure 8 depict the switch from host mode to onsite mode. Entering to onsite mode is always triggered by the debug exception upon the previously set breakpoint (Step 3). Thereafter, the trampoline removes the breakpoint immediately, as the breakpoint is no longer needed because the trampoline does not *jump* to the kernel code any more. The trampoline also calls the OASIS manager to prepare the vCPU and EPT for onsite mode.

[088] The first-time entrance requires the OASIS manager to set up the VMCS as well as the EPT as described above. The *CR3* register in the VMCS is cloned from the target thread's *CR3*. The *RIP* register is set to point to the first user-space instruction of the analyzer process in onsite mode.

[089] For returns following a VM Exit, the OASIS manager uses the context saved by the hardware to resume the analyzer thread. Besides that, it also checks whether the analyzer's paging structures in host mode are updated by the kernel. Exceptions such as page faults may entail address space changes. For new physical pages added to the paging structure or the analyzer's space, the OASIS manager updates the EPT accordingly.

[090] In the end, the trampoline uses VM-Entry instructions to switch the CPU to the VMX non-root mode and the hardware fetches the instruction pointed to by the *RIP* in the VMCS (step 4), which launches/resumes the analyzer in onsite mode.

[091] One possible security concern is that an adversary may reside in the target VM with the kernel privilege. If the target is a malicious software or program, the fundamental security mechanism deployed in OASIS is EPT-based access control. In one of the preferred embodiments of the present invention, the target address space is granted with read-only EPT permissions and the analyzer's code pages are protected as non-writable in onsite mode. Hence, the target cannot inject code into onsite mode. The design of OASIS allows for mapping manipulation attacks, because the untrusted paging structure is used in onsite mode. Any manipulation on mappings for the target's own address space is equivalent to data poisoning, as the consequence is that the analyzer reads the data chosen by the target. However, such an attack does not break cross-domain mapping consistency. Hence, the manipulation also takes effect on the adversary itself. A more insidious attack is to tamper with the PML4 entry storing the Grafting-Site GPA. By changing that entry to point to faked lower-level paging structures, the target can redirect the analyzer's control flow and data flow.

[092] Figure 9 depicts an example of a PML4 manipulation attack. The dashed arrow 901 shows the translation path in onsite mode before the attack, while the solid arrows 902 show the manipulated path. As depicted in Figure 9, it changes GPA_{λ} into GPA_m which is mapped to the physical page residing within the target VM (e.g., the faked PDPT page with #PFN 0x789). As a result, the MMU in onsite mode translates local VAs with the paging structures staged by the target, instead of those from the host OS. The attack maneuvers the analyzer to reads/write/execute at the page level. Nonetheless, the target is not able to modify the grafted PDPT page, for example Page Y in Figure 9, as well as all the subsequent PD and PT pages, because they are inaccessible from the target VM.

[093] It is difficult to entirely stall the manipulation attack because the PML4 page is fully under the target's control. Virtual address randomization is not helpful, since the target can map all virtual pages to the same GPA page. Nonetheless, due to memory paging, the target can only redirect the control and data flows at the page level, without the fine-grained control needed for sophisticated attacks such as Return-Oriented Programming (ROP) and Data-Oriented Programming (DOP).

[094] According to one of the preferred embodiments of the present invention, such an attack can be effectively detected. A page-fault exception will be raised with an overwhelmingly high probability when the hardware is about to fetch the very first instruction under the manipulated PML4 mapping. Since the analyzer's code pages are the only ones with the execute permission in onsite mode, the target must fake at least one correct VA-to-GPA mapping for the analyzer's code pages. As described above, GPAs of the analyzer code pages consist of random 36-bit page numbers (for 4KB pages). Consequently, the target is challenged to successfully guess a 36-bit secret when fabricating the VA-to-GPA mapping, at a low success probability of merely 2^{-36} .

Working Example

[095] As a working example, an implementation according to one of the preferred embodiments of the present invention is described below. In this working example, the OASIS is implemented on a PC with an Intel Core i5-4590 3.3 GHz processor (supporting VT-x) and 16 GB DRAM. It runs Linux kernel 3.13.0 with KVM for 64-bit x86-64 SMP. The target VM runs the same Linux version with 4 GB memory. The software of OASIS consists of 1327 Source lines of code (SLOC) for the OASIS manager, 312 SLOC for the trampoline, and 77 SLOC for the execve handler. This working example will be further described in detail below by separately describing Launch of Analyzer, OASIS Manager, Trampoline and Evaluation.

Launch of Analyzer

[096] In this working example, OASIS uses a wrapper program, a customized dynamic loader (hereinafter referred to as the OASIS loader), and a customized execve handler in the host OS to launch

and load the analyzer which is compiled into a PIE in advance. The first two components are in user space while the last is in the kernel.

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

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

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

OASIS Manager

[100] The OASIS manager is implemented as a kernel module, which allocates and configures the needed hardware and software infrastructure for onsite mode, including the VMCS and EPT. The Interrupt Descriptor Table (IDT) and Global Descriptor Table (GDT) are cloned from host mode. Configuration of the VMCS requires the knowledge of the target's `CR3` content, which there are several ways to achieve. One intrusive method is to insert a kernel module in the target OS which reports the `CR3`. A non-intrusive but more complicate method is to send an Inter-processor interrupt (IPI) to trap the target VM to the host OS. Using `CR3` saved in the VMCS, the analyzer can introspect the target kernel and locate *task struct* of the target, which contains the target's `CR3`.

[101] In this working example, PML4 entry corresponding to prefix `111111111` is chosen for grafting the analyzer paging structures, i.e. $\lambda t = 111111111$. The target VM's EPT could be updated at runtime by Linux KVM. To ensure the consistency at the EPT level, the OASIS manager hooks the KVM's EPT management routines so that any EPT update on the target is also replicated for onsite mode. The OASIS manager also sets the 32-bit Exception Bitmap in the VMCS to trap all exceptions into the

trampoline. Exceptions (including page faults) and interrupts are directly trapped. Since no kernel runs in onsite mode, a system call triggers a page fault exception, which is also trapped to the trampoline.

Trampoline

5

[102] In this working example, the trampoline is implemented as a kernel module, which is hooked to the host OS's debug exception handler and the VM Exit handler. Proper filters can be added so that the trampoline only handles OASIS related debug exceptions and VM Exits.

10 [103] **VM Exit.** The trampoline uses data stored in the VMCS of the onsite mode environment to identify the event type (system call, page fault and general exception/interrupt). It prepares the context for the host OS accordingly, including the kernel stack for the analyzer as well as all relevant general and control registers in the physical CPU. For system calls, it copies *RAX*, *RDI*, *RSI*, *RDX*, *R10*, *R8*,
15 *R9* registers from the VMCS to the physical registers, where *RAX* contains the system call number and others store parameters (if any). For general exceptions, it pushes *RIP* in the VMCS to the kernel stack. In case of page fault, it also copies *CR2* of the vCPU to the physical *CR2*, which is supposed to store the faulting linear address. The trampoline uses the processor's debug registers to set a breakpoint, which is execution-triggered and local to the analyzer task. The breakpoint is set at the virtual address of the instruction which the analyzer resumes execution. For exceptions, it uses the address in *RIP*
20 which points to the faulting instruction; for system calls, it uses the address in *RCX* which points to the instruction next to the system call instruction. Since Linux does not use the processor's task switch facility, the local breakpoint is in fact valid for all processes in the host. Nonetheless, the likelihood of being triggered by other processes remains slim, because the 512 GB linear address at the base 0x7F8000000000 is rarely used by other processes. In the end, the trampoline uses a *jump* instruction
25 to pass the control to the kernel exception/interrupt/system call handler. The kernel handler locates its stack based on the Per-processor Data Area (PDA) and processes the event as if it originates in the analyzer's userland.

[104] **Debug Exception.** Once the debug exception is triggered due to the preset breakpoint, the
30 trampoline's task is to start or resume the analyzer thread in onsite mode. With the hardware's Interrupt Stack Table (IST), the trampoline handles the debug exception with a different stack from the analyzer's kernel stack. The interrupt stack is also the one used during handling VM exit. The kernel stack used for handling the analyzer events is not contaminated by the trampoline execution. Note that the breakpoint is triggered after *iret* or *sysret* in the OS's handler is executed successfully. Hence, from the
35 kernel point of view, the thread has entered user mode. The execution of the trampoline after the debug exception, which is still under the analyzer thread's context, is oblivious to the rest of kernel. Hence, entering or exiting onsite mode preserves the analyzer thread context in the OS's view.

[105] **System Calls.** Generally, the trampoline is able to support all system calls. Due to the virtual
40 address relocation in onsite mode, the trampoline needs to understand the semantics of system calls.

This working example is set to support at least 47 commonly used Linux system calls, including those for memory management, file system, network I/O management and signals, which may be related/needed for the implementation of this invention. They also include some system related system calls such as *arch_prctl*, which sets the base of FS segment.

5

Evaluation

[106] In this working example, the evaluation process entails the first step of verifying whether the host OS views the analyzer as a regular thread though it runs in onsite mode. In the test, the launcher program issues *getpid* in the host while the analyzer issues *getpid* in onsite mode. The result shows that the launcher program's Process ID (PID) is 4617 while the analyzer gets 4618. It corroborates the correctness of OASIS as the analyzer in onsite mode is indeed the child process of the launcher.

10

[107] Although the analyzer in onsite mode receives the same service as in host mode, its system calls take longer time than in a native environment due to the overhead of OASIS mode switches. Therefore, to test the actual performance drop, the time of execution of five commonly used system calls in onsite mode and normal user mode is measured; the result is shown in Table 1 below, which shows that the average overhead of system calls in onsite mode is about 2.21 μ s.

15

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

20

Table 1: Popular system call time in two modes (in μ s)

[108] Further, to illustrate the characteristics of OASIS's performance, the time cost of hardware events during the mode switch is tested. A round-trip mode switch entails at least three hardware events: a VM Exit, a VM Entry and a debug exception, as shown in Table 2 below. Therefore, it is about 63% (1.39 μ s / 2.21 μ s) of total system call overhead.

25

	Onsite-to-host	Host-to-onsite
VM Exit	0.387	-
VM Entry	-	0.512
Debug exception	-	0.493
SUM	1.39	

Table 2: Hardware time costs in mode switches (in μ s)

[109] Besides, OASIS is also tested with three applications: SVM2, a program using OpenSSL library to encrypt and decrypt files, and a network program to evaluate the overhead of program loading and

30

the correctness of system call support. According to the results, it takes about 20.31 milliseconds, 32.27 milliseconds and 16.39 milliseconds to launch these applications, respectively. The dominant cost contains two parts. One is to initialize the needed VMCS structure and EPT; and the other is to handle the system services requested by the loader to find and map shared libraries. The above tests

5 show that OASIS correctly supports file operations and network I/O.

[110] Hereinafter described are three examples of applications of OASIS, which describes how the OASIS may be used and the advantages of using OASIS. These applications demonstrate that OASIS is handy to use and is useful for extracting intelligence from target's address space with little runtime

10 overhead.

Application Example I - Dynamic CFG Generation

[111] The first example of application of OASIS uses a static code analysis tool to dynamically

15 generate the control flow graphs (CFG) of the Just-In-Time (JIT) binary code emitted by the Chakra Javascript engine. An existing tool is used to generate CFGs in real time for binaries emitted by the JIT compiler.

[112] A JavaScript (JS) code is interpreted and executed in a confined environment which restricts

20 its behaviors. Hence, a malicious JS usually relies on the JIT code to realize its malicious goals. Since memory vulnerabilities allow a JS malware to trick the JIT compiler to emit arbitrary code, it is desirable to analyze the JIT code at runtime.

[113] It is appreciated that CFG generation is a mature static analysis technique. Most tools take as

25 input either an executable and linkable format (ELF) file or a memory image file. The crux of CFG generation is to interpret the symbols and the virtual addresses used in the target code's direct transfers so that an edge may be generated to link two basic blocks. Typically, the tool loads the file into its own address space and creates a mapping between the symbols or virtual addresses of the target and their actual memory locations in the tool's address space. To avoid the confusion between this mapping and

30 the VA-to-PA address mapping, the former is referred to as location mapping. Hence, without the support of OASIS, it is infeasible for a static analysis tool to generate the CFG for the JIT code. Because of the address space barrier, the tool cannot build the location mappings. Existing techniques have to rely on the JIT compiler to generate the CFG on the fly. As a result, it inflates the compiler and makes it more vulnerable to attacks.

[114] OASIS offers an alternative solution. Although the JIT code resides in the JS engine's address

35 space, OASIS can load the CFG generation tool into the same address space. Hence, the location mapping becomes an identity map. In onsite mode where the tool's address space is grafted into the JS engine's, the virtual address appearing in the JIT code may be directly used by the tool to fetch the

40 code blocks.

[115] In this application example I, Dyninst is chosen as the CFG generator, and ChakraCore is chosen as the JS engine. A hook is placed in Chakra's encoding function, which converts IR into the native code. When a piece of new JIT code is generated, the JIT compiler reports its base address and the size. Since the JIT code is compiled from the JS method, its code always occupies a continuous
 5 VA region. Therefore, it is sufficient to get the base address and the size. No physical address is needed because Dyninst running in onsite mode directly uses the JIT code's VA-to-PA mappings.

[116] The default Dyninst binary does not support memory-based CFG generation. In this application example I, a few functions needed for memory image based CFG generation is supplied following the
 10 instructions on its official web site. Among these functions, the key one is to provide the location mapping for its code parsing. In Dyninst, the *CodeSource* interface is used by *ParseAPI* to retrieve binary code from a binary code object. *ParseAPI* provides a default implementation based on *SymtabAPI* that supports many common binary formats including ELF, Common Object File Format (COFF), and Portable Executable (PE) format. A set of member methods in class *CodeSource*
 15 is implemented to support parsing binary code in memory. After constructing a *CodeSource* instance for the newly generated JIT code, *parse()* function is invoked to analyze the binary. In addition, the input interface of Dyninst is also adapted. The new interface reads the base address and code size reported by the JIT compiler directly from Chakra's address space.

[117] The tool is tested with the JetStream Javascript benchmark and a hand-made malicious JS that emulates the attack in Chakra JIT CFG Bypass to inject binaries to the JIT code. On average, Dyninst takes about 767 microseconds to parse each JIT code. Table 3 below shows some basic information about the generated CFGs.

	JetStream	Malicious JS
# of JIT code	20	1
Average Size (bytes)	0x686	0x275
# of Basic Blocks	1665	29
# of Edges	2290	42

Table 3: Results of CFG generation

[118] In this malicious JS test, the attacking code inserted into the JIT code is successfully found. The tool of this application example I still trusts the JS engine to supply the base address and code size of the emitted code. It could be enhanced to generate the CFG without that dependence. One possible
 30 approach is to monitor the *mprotect* system call that is used to change the permission of the newly generated code buffer from non-executable to executable.

[119] The CFG generation tool shows that onsite mode breaks the address space barrier between two processing by fusing them into one, without compromising security. The analyzer can conveniently

reference the target memory by using the target's virtual address directly with ensured mapping consistency. It shows that OASIS can successfully run an application in onsite mode to share the virtual address space with the target program.

5 **Application Example II - Virtual Address Space Reconnaissance**

[120] The second example of application of OASIS delineates the target's virtual address space without relying on any target kernel objects and then analyzes the runtime data on its heap. Hence, it could be considered as a memory reconnaissance tool. In this application example II, a malware is used as the target.

[121] It is appreciated that the target's virtual address space is the basis for high-level analysis. Existing approaches typically follow the conventional address space layout or trust the virtual memory area (VMA) structures provided by the target kernel. Both methods fail to work when the target reshapes its address space or subverts the target kernel. In this application example II, the reconnaissance tool is developed on top of OASIS to plot the layout of the target thread's live address space without trusting the target kernel or assuming any prior knowledge. The tool also discovers potential pointers in the virtual memory and their values.

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

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

[124] There are several alternative ways to obtain the starting page. One straightforward choice is to choose a random page or choose one from the default layout. However, better options are to select the pages based on the current execution context. One of such options is the target's stack page. When the target VM is trapped to the hypervisor, its stack pointer (*RSP*) value may be read from its VMCS structure. Since stack frames contain return addresses, the program also finds out the code

pages that are currently in execution. Another one of such options is to find possible virtual addresses stored in general registers during VM exit or the VA of the intercepted instruction.

[125] Those skilled in the art would know that range checking is a popular method used to decide whether the given eight bytes are a valid VA or not. They are compared against the known VA ranges whose information is extracted from the target kernel. As compared to the range-checking approach, the pointer-test approach is much faster for genuine pointers. However, it costs more CPU cycles for non-pointers due to the exceptions.

[126] This application example II is optimized to minimize the cost of exceptions. The first optimization is to filter out non-canonical addresses, namely between 0x7FFFFFFFFFFFFF and 0xFFFF800000000000, which are considered as illegal by the hardware implementing 48-bit addresses. The second optimization is a three-pronged approach, which reduces the overhead of handling page faults triggered by accessing unmapped addresses. Firstly, the onsite mode GDT is configured so that the tool runs with Ring 0 privilege, instead of Ring 3. It avoids expensive privilege switches when page faults are thrown out. Secondly, a special page fault handler is installed for the tool. The handler's main task is to set a flag and advances the test according to the step length. The IDT is configured to disable stack switches for page faults. Thirdly, different from normal handlers that use *iret* to resume the faulting instruction, this application example II uses *ret* instruction to make faster control transfer. The total implementation consists of 446 SLOC.

[127] For the purpose of testing this application example II, the address space reconnaissance tool is applied upon Apache and Ustealer (<https://github.com/atmoner/Ustealer>) which is a malware stealing Ubuntu information. Starting from the stack page, the tool gradually discovers the target's virtual pages and all potential pointers therein. The target kernel's report on the address space is considered as the baseline. The number of mapped pages is counted by checking the page table, and the mapped pages is calculated using the VMA structures in the target kernel. The results from both experiments are summarized in Table 4 where the VMA-based page counts are indicated in brackets.

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

Table 4: Apache and Ustealer's address spaces

[128] As shown in Table 4, the tool identifies most pages which do have mappings in the page tables, and more VMAs than reported by the target kernel due to the target kernel's lazy memory allocation. For the sake of better performance, the target kernel may allocate the virtual address space without

mapping all physical pages, which lead to discontinued VMAs. The pointer test results hence reflect the actual mappings defined in the page tables, which are more fine-grained and more accurate. As shown in Table 4, in the test with Ustealer, the target kernel allocated 4928 pages while only 909 pages are actually mapped and only 13 VMA regions are reported. This indicates that the malware aggressively requests the target kernel to allocate large chunks of memory. In addition to the above results, the measurement in this application example II also shows that the page fault handler execution takes about 194 nanoseconds, and it takes the tool about 26 milliseconds and 40 milliseconds to explore Ustealer and Apache, respectively. Therefore, the main overhead is due to page faults induced by non-pointers in the pages.

[129] Further, an analysis is conducted upon the 287 (potential) pointers discovered on one 4 KB stack page of Apache, which accommodates up to 512 8-byte pointers. According to the pointer values, they are grouped into four categories: heap pointers (i.e., those pointing to the heap object), stack pointers, library pointers and Apache pointers (i.e., those pointing to Apache's code and data sections). Figure 10(a) is a graphic representation of the four categories, which shows a seemingly flat line formed by Apache pointers. Figure 10(b) visualizes this segment and shows that they are 49 adjacent pointers pointing to adjacent memory locations in Apache's data section, which shows a high probability that they are an array of pointers pointing to another array.

[130] The tool in this application example II also attempts to discover possible link-lists using a chain of pointers in the heap. The analysis is based on the pattern that pointers in link-list nodes always have the same offset to the respective node's address. The results are shown in Table 5 below, where the third column is the distance between the pointer's address and its node address, and the last column is the number of pointers pointing to the head of the list.

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

Table 5: Potential link-lists in one of Apache's heap pages

[131] The above results show that OASIS empowers the analysis tool to conduct large-scale on-demand non-intrusive virtual memory accesses, without trusting the target OS. OASIS efficiently plots the virtual address space layout of a running target without relying on memory dump or the target kernel VMA structures; it not only saves the hassle of memory dump or software-based address translation, but also helps obtaining the address-related semantics, which are useful to understand the target's binary level behaviors.

[132] It will be appreciated that the OASIS may be further enhanced when combined with static and dynamic analysis tools, of which one of the advantages is that it provides a holistic study on the target, such as a malware.

5 **Application Example III - System Call Monitoring**

[133] The third example of application of OASIS is a system call monitoring tool that detects target's system call issuance. Due to its non-intrusiveness, OASIS does not have the capability of intercepting runtime events. Nonetheless, almost all executions leave traces in the memory. Hence, by analyzing
10 the traces, the events may be inferred. This application example III is designed based on this observation to show OASIS's capability of detecting system events in a target.

[134] The kernel and the hardware have predefined memory write operations on the kernel stack when receiving system calls. Hence, the concept of this application example III is to continuously
15 monitor the virtual address(s), which the kernel is expected to write in the stack. Figure 11 depicts the evolution of the user program's kernel stack during a system call. Whenever the target program runs in user-mode, the process's kernel stack is empty. In response to the *syscall* instruction, the kernel always first pushes the user-mode stack segment and stack pointer into the kernel stack, followed by other information including the system call number, the return address and parameters. When the
20 system call handler completes its service, it pushes RAX storing the system call return value into the stack, before firing the *sysret* instruction. In the end, the kernel stack pointer is restored to its initial position. Note that the contents of the stack are not zeroed.

[135] It is not sufficient to monitor the return address alone. Most system calls are issued from the C
25 library functions. Hence, the return address only reflects which libc function is called, without the information of the higher-level caller in the application code. In contrast, *RSP* is associated with the runtime status of the application which is more likely to be different in each system call. Therefore, to monitor a system call issuance, the tool monitors the stack placeholder for the user-mode *RSP* and the return address.

[136] Generally, there are two possible ways to locate the kernel stack for the target thread. One
30 approach is to use the content of *RSP* when the target VM is trapped to the host from the target thread's kernel mode. Its advantage is that it is ensured to be the genuine one used by the target. Nonetheless, it fails when the trap is from user mode or other processes. Another approach is to introspect the target
35 kernel to obtain the *thread_info* structure of the target program to locate the stack, which is used in this application example III.

[137] The monitor is essentially a loop that continuously reads the user-mode stack pointer image in
40 the kernel stack. It compares the outcome with its previous read. If they are not equal, the tool saves the user-mode *RSP*, *RIP*, *RAX* images, six *syscall* arguments, the *syscall* return value, and *EFLAGS*.

The total implementation consists of 121 SLOC. It takes about 20 CPU cycles to record one event, which is speedy enough to catch the kernel processing of most system calls.

[138] The monitoring tool in this application example III also captures exceptions and interrupts because these events apparently entail the same kernel stack evolution. In those cases, the interpretation of system call behavior fails due to the inappropriate system call number and/or parameters. It is appreciated that the noise is a double-edged sword. Although it may cause errors in system call analysis, it provides additional intelligence to understand the target software behavior. For instance, frequent non-present page faults indicate the target's intensive memory usage and frequent interrupts may indicate intensive I/O operations.

[139] Differentiating system calls, interrupts and exceptions mandate deeper analysis of the stack content. One indicator is the *RF* bit (i.e. the 16th bit) in RFLAGS register image stored in the stack. For any fault-class exceptions, the hardware sets this bit in the register's stack image; otherwise, it is cleared. Therefore, it is an accurate indicator to differentiate fault-class exceptions and system calls.

[140] The tool in this application example III is tested against four unnamed malware samples downloaded from Github (<https://github.com/greg5678/Malware-Samples>) and two benign programs (Firefox and Bash). The malware samples are 32-bit Linux executables using the old-fashioned *int 80* method to issue system calls. Nonetheless, the 64-bit kernel stack still evolves as described above. During the test, monitor stops after catching 100 events, and system calls recorded by *strace* in the target as the baseline is used to evaluate effectiveness of monitoring. The results are shown in Table 6 below, where M1 to M4 denote four unnamed malware samples.

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

Table 6: Results of system call monitoring

[141] As shown in Table 6, the tool in this application example III captures all system calls from Firefox and Bash, but misses a significant percentage of system calls from M2 and M4. After manual investigation, it is found that both malwares continuously issue system calls from the same calling site, which does not change the user-stack pointer stored in the kernel stack. Therefore, the tool may be enhanced by taking into account of the kernel stack placeholders for system call numbers and user-mode RIP. The tool can also be enhanced by leveraging Intel TSX transactional memory so that the target kernel's write accesses at that location may be captured, no matter whether the content is changed or not.

[142] As the purpose of this application example III is to show OASIS's capability of monitoring, the captured sequence of the system calls are not analyzed. However, a simple frequency analysis reveals that the first sample makes intensive network operations as it issues more *sys_socketcall* than other system calls while the fourth sample requests *sys_write* more often, and all of them copy files in */usr* directory.

[143] Therefore, this application example III demonstrates that non-intrusive memory monitoring can help to infer certain system events without intercepting them with the knowledge of memory evolution patterns, which means OASIS has certain capability of inferring events in the target VM by monitoring memory content evolution. Further, it is worth to note that by combining OASIS with the transactional memory, the capability of event monitoring of OASIS may be further enhanced.

[144] According to the aforementioned description of the present invention, working example and application examples, one would appreciate that OASIS-based analysis is a dynamic analysis technique since it inspects a running software. However, different from existing schemes, it focuses on the target software's virtual memory, instead of directly upon runtime activities. It provides a broader scope of intelligence collection and may be used to guide behavior analysis, which is often implemented in an intrusive manner.

[145] In addition, OASIS-based analysis is different from analysis techniques which use hardware virtualization techniques to trap and intercept malware operations. It does not necessarily require intercepting malware execution. In fact, one of its advantages is its non-intrusiveness. Although malware may determine that it is presently running in a virtual machine, it cannot decide the presence of the analyzer (unless by using side-channel attacks). Considering the increasingly deployment of virtual machines and cloud-hosted servers, malware may still activate its attack. Nonetheless, when intrusiveness is not a concern, OASIS-based analysis can conveniently be combined with Ether.

[146] Further, OASIS-based analysis is also different from existing memory analysis techniques which require a copy of the physical memory of the target. There are at least two shortcomings in the existing memory analysis techniques. Firstly, the analyzer tool cannot use the real-time address mappings to translate virtual addresses to physical addresses, spawning a thorny semantic-gap problem. Secondly, it costs time to take memory snapshots; and once a copy is taken, it may become stale as it is no longer updated by the target. Therefore, the existing techniques are more suitable for post-mortem analysis. However, according to the aforementioned details, OASIS ensures mapping consistency and the data fetched on-demand is ensured to be fresh.

[147] An OASIS-based introspection system is an out-of-VM introspection system, which allows a monitor program outside of the target VM to inspect the target VM's kernel states. As compared to in-VM introspection schemes which rely on a trusted module in the target kernel, out-of-VM introspection resists stronger attacks and are easier to deploy.

[148] It is therefore appreciated based on the above that, the applications of the OASIS in the present invention include but not limited to: detecting a potential target program, analyzing a target program's live address space; allowing an analyzer to securely execute in onsite mode wherein its local address space is fused with the target's virtual address space, without disrupting the target execution, but still runs in the same fashion as in a normal environment with the capability of receiving OS services; real-time monitoring; live forensics and virtual machine (VM) introspection etc. OASIS has the potential to foster new dynamic analysis methods/systems to exploit the virtual address space. For instance, an SVM classifier could run on top of OASIS to learn and extract address space patterns. It is easy to apply OASIS, and with the support of OASIS the analysis applications can make intensive on-demand accesses to the target's address space at native speed with guaranteed mapping consistency.

[149] While the disclosure contained herein has set forth some of the preferred embodiments of the present invention, working example and application examples of OASIS, it is not intended to limit that the OASIS can only be implemented/applied/used as in these embodiments/examples. It will be appreciated by those skilled in the art that variations to the combination of elements and steps disclosed may be made without departing from the scope and spirit of the invention. Such variations may include, but are not limited to, selection of alternate platforms and operating systems, and implementation of the process as a servlet or other program embodiment.

CLAIMS

1. A computer-implemented method for detecting, analyzing, monitoring, investigating and/or executing a target program, comprising:
 - 5 - obtaining a tool paging structure of a tool, wherein the tool paging structure is created by a host operating system;
 - obtaining a target paging structure of the target program, wherein the target paging structure is created by a target virtual machine; and
 - 10 - generating a hybrid paging structure by grafting at least one part of the tool paging structure into at least one part of the target paging structure; wherein the hybrid paging structure is used for virtual address translation when detecting, analyzing, monitoring, investigating and/or executing the target program using the tool.
2. The computer-implemented method according to claim 1, wherein the tool appears as a normal

15 application to the host operating system when receiving service from the host operating system and accesses a virtual address space of the target program with the same paging structure as the target program when detecting, analyzing, monitoring, investigating and/or executing the target program.
3. The computer-implemented method according to claim 1 or 2, wherein the target program is a

20 malware or a benign software.
4. The computer-implemented method according to any of claims 1 to 3, wherein the tool comprises at least a static analysis tool and/or at least a dynamic analysis tool.
- 25 5. The computer-implemented method according to any of claims 1 to 4, wherein the tool detects, analyzes, monitors, investigates and/or executes the target program at runtime.
6. The computer-implemented method according to any of claims 1 to 5, wherein the tool is a

30 position-independent executable.
7. The computer-implemented method according to any of claims 1 to 6, wherein when the hybrid

paging structure is used for virtual address translation, different access permissions are selected to be set on pages storing the target program's code and data according to:
 - level of security required at the time of translating,
 - 35 - trust level of the pages storing the target program's code and data, and
 - the tool's needs.
8. The computer-implemented method according to any of claims 1 to 7, wherein when the hybrid

40 paging structure is used for virtual address translation, execute permission is set on pages storing the tool's code.

9. The computer-implemented method according to any of claims 1 to 7, wherein when the hybrid paging structure is used for virtual address translation, at least one of read permission and write permission is selected to be set on pages storing the tool's code according to:

- level of security required at the time of translating, and
- trust level of the pages storing the tool's code.

10. A system for detecting, analyzing, monitoring, investigating and/or executing a target program, comprising:

- a host operating system;
- a target virtual machine; and
- a tool;

wherein a tool paging structure is provided to the tool by the host operating system;

wherein a target paging structure is provided to the target program by the target virtual machine;

wherein a hybrid paging structure is generated by grafting at least one part of the tool paging structure into at least one part of the target paging structure; and

wherein the hybrid paging structure is used for virtual address translation when detecting, analyzing, monitoring, investigating and/or executing the target program using the tool.

11. The system according to claim 10, wherein the tool appears as a normal application to the host operating system when receiving service from the host operating system and accesses a virtual address space of the target program with the same paging structure as the target program when detecting, analyzing, monitoring, investigating and/or executing the target program.

12. The system according to claim 10 or 11, wherein the target program is a malware or a benign software.

13. The system according to any of claims 10 to 12, wherein the tool comprises at least a static analysis tool and/or at least a dynamic analysis tool.

14. The system according to any of claims 10 to 13, wherein the system further comprises transactional memory.

15. The system according to any of claims 10 to 14, wherein the tool detects, analyzes, monitors, investigates and/or executes the target program at runtime.

16. The system according to any of claims 10 to 15, wherein the tool is a position-independent executable.

17. The system according to any of claims 10 to 16, wherein when the hybrid paging structure is used for virtual address translation, different access permissions are selected to be set on pages storing the target program's code and data according to:

- level of security required at the time of translating,
- trust level of the pages storing the target program's code and data, and
- the tool's needs.

18. The system according to any of claims 10 to 17, wherein when the hybrid paging structure is used for virtual address translation, execute permission is set on pages storing the tool's code.

19. The system according to any of claims 10 to 17, wherein when the hybrid paging structure is used for virtual address translation, at least one of read permission and write permission is selected to be set on pages storing the tool's code according to:

- level of security required at the time of translating, and
- trust level of the pages storing the tool's code.

20. A computer program comprising instructions which, when the program is executed by a computer, cause the computer to carry out the method according to any one of claims 1 to 9.

21. A computer-readable medium comprising instructions which, when executed by a computer, cause the computer to carry out the method according to any one of claims 1 to 9.