

**Systems And Methods for Exposing A Current Processor Instruction
Upon Exiting A Virtual Machine**

RELATED APPLICATIONS

5 [0001] This application claims the benefit of the filing date of U.S. provisional patent application No. 62/038,476, filed on Aug. 18, 2014, entitled "Systems And Methods for Exposing A Current Processor Instruction Upon Exiting A Virtual Machine", the entire contents of which are incorporated by reference herein.

BACKGROUND

10 [0002] The invention relates to computer security, and in particular to performing computer security operations in hardware virtualization configurations.

[0003] Malicious software, also known as malware, affects a great number of computer systems worldwide. In its many forms such as computer viruses, worms, rootkits, and spyware, malware presents a serious risk to millions of computer users, making them vulnerable to loss of data and
15 sensitive information, identity theft, and loss of productivity, among others.

[0004] Modern computing applications often employ hardware virtualization technology to create simulated computer environments known as virtual machines (VM), which behave in many ways as physical computer systems. In applications such as server consolidation and infrastructure-as-a-service, several virtual machines may run simultaneously on the same
20 computer system, sharing the hardware resources among them, thus reducing investment and operating costs. Each virtual machine may run its own operating system and/or software, separately from other virtual machines. Due to the steady proliferation of computer security threats such as malware and spyware, each such virtual machine potentially requires protection.

[0005] Some security solutions protect a virtual machine by monitoring the manner in which
25 guest processes executing within the protected VM access memory, to identify potential malicious activity. In one example, a computer security program may configure the processor to generate an internal event (e.g., an exception or a VM exit event) when an attempt is made to write to, or execute code from, a specific region of memory, e.g. a region of memory used by a guest process. Such processor events typically suspend the execution of the current thread and

switch the processor to executing an event handler routine, which may form part of the computer security program. The computer security program may thus detect an attempt to access memory in a manner which may be indicative of malware. After analyzing the event, the computer security program may emulate the processor instruction which was under execution when the event occurred, and may return execution to the original thread. Such methods are generically known in the art as trap-and-emulate.

[0006] Conventional trap-and-emulate methods may place a substantial computational burden on the host computer system, potentially impacting user experience and productivity. Therefore, there is considerable interest in developing efficient computer security systems and methods suitable for virtualization environments.

SUMMARY

[0007] According to one aspect, a host system comprises at least one hardware processor configured to execute a virtual machine and a computer security program. The at least one processor is further configured to determine whether executing a guest instruction within the virtual machine causes a violation of a memory access permission. In response, when executing the guest instruction causes the violation, the at least one hardware processor is further configured to write a part of a machine code representation of the guest instruction to a predetermined location accessible to the computer security program, to suspend the execution of the guest instruction, and in response to suspending the execution of the guest instruction, switch to executing the computer security program, wherein the computer security program is configured to determine whether the violation is indicative of a computer security threat.

[0008] According to another aspect, a method of protecting a host system from computer security threats, the method comprises, in response to receiving a guest instruction for execution, employing at least one hardware processor of the host system to determine whether executing the guest instruction causes a violation of a memory access permission, wherein the guest instruction executes within a virtual machine exposed by the host system. The method further comprises, in response to determining whether the guest instruction causes the violation, when executing the guest instruction causes the violation, employing the at least one hardware processor to write a part of a machine code representation of the guest instruction to a predetermined location

accessible to the computer security program, and employing the at least one hardware processor to suspend the execution of the guest instruction. The method further comprises, in response to suspending the execution of the guest instruction, switching to executing a computer security program configured to determine whether the violation is indicative of a computer security threat.

[0009] According to another aspect, at least one hardware processor of a host system is configurable, in response to receiving a guest instruction for execution, to determine whether executing the guest instruction causes a violation of a memory access permission, wherein the guest instruction executes within a guest virtual machine exposed by the host system. The at least one hardware processor is further configurable, in response to determining whether the guest instruction causes the violation, when executing the guest instruction causes the violation, to write a part of a machine code representation of the guest instruction to a predetermined location accessible to the computer security program, to suspend the execution of the guest instruction, and in response to suspending the execution of the guest instruction, to switch to executing a computer security program, wherein the computer security program is configured to determine whether the violation is indicative of a computer security threat.

[0010] According to another aspect, a non-transitory computer-readable medium stores instructions which, when executed by at least one hardware processor of a host system, cause the host system to form a computer security program configured to determine whether a violation of a memory access permission is indicative of a computer security threat. The at least one hardware processor is configurable, in response to receiving a guest instruction for execution, to determine whether executing the guest instruction causes the violation, wherein the guest instruction executes within a virtual machine exposed by the host system. The at least one hardware processor is further configurable, in response to determining whether the guest instruction causes the violation, when executing the guest instruction causes the violation, to write a part of a machine code representation of the guest instruction to a predetermined location accessible to the computer security program, to suspend the execution of the guest instruction, and in response to suspending the execution of the guest instruction, to switch to executing the computer security program.

BRIEF DESCRIPTION OF THE DRAWINGS

[0011] The foregoing aspects and advantages of the present invention will become better understood upon reading the following detailed description and upon reference to the drawings where:

[0012] Fig. 1 shows an exemplary hardware configuration of a host computer system according to some embodiments of the present invention.

[0013] Fig. 2-A shows an exemplary set of virtual machines exposed by a hypervisor executing on the host system, and a computer security module (CSM) protecting the set of virtual machines according to some embodiments of the present invention.

[0014] Fig. 2-B shows an alternative embodiment of the present invention, wherein a CSM executes below a virtual machine, and wherein an exception handler executes within the protected virtual machine.

[0015] Fig. 2-C shows yet another embodiment of the present invention, wherein both the CSM and the exception handler execute within the protected virtual machine.

[0016] Fig. 3 shows an exemplary configuration of virtualized hardware exposed as a guest virtual machine according to some embodiments of the present invention.

[0017] Fig. 4 shows a set of exemplary memory address translations in a hardware virtualization configuration as shown in Fig. 2-A, according to some embodiments of the present invention.

[0018] Fig. 5 shows exemplary components of a processor according to some embodiments of the present invention.

[0019] Fig. 6 shows an exemplary suspend event register of the processor according to some embodiments of the present invention.

[0020] Fig. 7 shows an assembly language representation of an exemplary processor instruction of the x86 instruction set, and its corresponding machine code representation.

[0021] Fig. 8 shows an exemplary sequence of steps performed by the processor to execute a processor instruction according to some embodiments of the present invention.

[0022] Fig. 9 illustrates an exemplary sequence of steps performed by a computer security module to protect a guest virtual machine according to some embodiments of the present invention.

DETAILED DESCRIPTION OF PREFERRED EMBODIMENTS

[0023] In the following description, it is understood that all recited connections between structures can be direct operative connections or indirect operative connections through intermediary structures. A set of elements includes one or more elements. Any recitation of an element is understood to refer to at least one element. A plurality of elements includes at least two elements. Unless otherwise required, any described method steps need not be necessarily performed in a particular illustrated order. A first element (e.g. data) derived from a second element encompasses a first element equal to the second element, as well as a first element generated by processing the second element and optionally other data. Making a determination or decision according to a parameter encompasses making the determination or decision according to the parameter and optionally according to other data. Unless otherwise specified, an indicator of some quantity/data may be the quantity/data itself, or an indicator different from the quantity/data itself. A computer program is a sequence of processor instructions carrying out a task. Computer programs described in some embodiments of the present invention may be stand-alone software entities or sub-entities (e.g., subroutines, libraries) of other computer programs. Unless otherwise specified, a computer security program is a computer program that protects equipment and data against unintended or unauthorized access, modification or destruction. Unless otherwise specified, a process is an instance of a computer program, such as an application or a part of an operating system, and is characterized by having at least an execution thread and a virtual memory space assigned to it, wherein a content of the respective virtual memory space includes executable code. Unless otherwise specified, a page represents the smallest unit of virtual memory that can be individually mapped to a physical memory of a host system. The term "logic" encompasses hardware circuitry having a fixed or a reconfigurable functionality (e.g., field-programmable gate array circuits), but does not encompass software emulating such functionality on a general-purpose computer. Unless

otherwise specified, a register represents a storage component integrated with or forming part of a processor, and distinct from random-access memory (RAM). Computer readable media encompass non-transitory media such as magnetic, optic, and semiconductor storage media (e.g. hard drives, optical disks, flash memory, DRAM), as well as communication links such as
5 conductive cables and fiber optic links. According to some embodiments, the present invention provides, *inter alia*, computer systems comprising hardware (e.g. one or more processors) programmed to perform the methods described herein, as well as computer-readable media encoding instructions to perform the methods described herein.

[0024] The following description illustrates embodiments of the invention by way of example
10 and not necessarily by way of limitation.

[0025] Fig. 1 shows an exemplary hardware configuration of a host system 10 according to some embodiments of the present invention. Host system 10 may represent a corporate computing device such as an enterprise server, or an end-user device such as a personal computer, tablet computer, or smartphone. Other exemplary host systems include TVs, game consoles, wearable
15 computing devices, or any other electronic device having a memory and a processor. Host system 10 may be used to execute a set of software applications, such as a browser, a word processing application, and an electronic communication (e.g., email, instant messaging) application, among others. In some embodiments, host system 10 is configured to support hardware virtualization and to expose a set of virtual machines, as shown below.

[0026] Fig. 1 illustrates a computer system; the hardware configuration of other host systems, such as smartphones and tablet computers, may differ. System 10 comprises a set of physical devices, including a processor 12, a memory unit 14, a set of input devices 16, a set of output devices 18, a set of storage devices 20, and a set of network adapters 22, all connected by a controller hub 24. In some embodiments, processor 12 comprises a physical device (e.g. multi-
25 core integrated circuit formed on a semiconductor substrate) configured to execute computational and/or logical operations with a set of signals and/or data. In some embodiments, such logical operations are delivered to processor 12 in the form of a sequence of processor instructions (e.g. machine code or other type of software). Some embodiments of the present invention introduce changes to the structure and functionality of a conventional processor, the

respective changes enabling processor **12** to operate more efficiently in hardware virtualization configurations.

[0027] Memory unit **14** may comprise volatile computer-readable media (e.g. RAM) storing data/signals accessed or generated by processor **12** in the course of carrying out instructions.

5 Input devices **16** may include computer keyboards, mice, and microphones, among others, including the respective hardware interfaces and/or adapters allowing a user to introduce data and/or instructions into host system **10**. Output devices **18** may include display devices such as monitors and speakers, among others, as well as hardware interfaces/adapters such as graphic cards, allowing host system **10** to communicate data to a user. In some embodiments, input
10 devices **16** and output devices **18** may share a common piece of hardware, as in the case of touch-screen devices. Storage devices **20** include computer-readable media enabling the non-volatile storage, reading, and writing of processor instructions and/or data. Exemplary storage devices **20** include magnetic and optical disks and flash memory devices, as well as removable media such as CD and/or DVD disks and drives. The set of network adapters **22** enables host
15 system **10** to connect to a computer network and/or to other devices/computer systems. Controller hub **24** generically represents the plurality of system, peripheral, and/or chipset buses, and/or all other circuitry enabling the communication between processor **12** and devices **14**, **16**, **18**, **20** and **22**. For instance, controller hub **24** may include a memory management unit (MMU), an input/output (I/O) controller, and an interrupt controller, among others. In another example,
20 controller hub **24** may comprise a northbridge connecting processor **12** to memory **14** and/or a southbridge connecting processor **12** to devices **16**, **18**, **20**, and **22**. In some embodiments, parts of controller hub (such as the MMU) may be integrated with processor **12**, i.e., may share a common substrate with processor **12**.

[0028] Fig. 2-A shows an exemplary functional configuration according to some embodiments
25 of the present invention, wherein host system **10** uses hardware virtualization technology to operate a set of guest virtual machines **52a-b** exposed by a hypervisor **50**. Such configurations are common in applications such as cloud computing and server consolidation, among others. A virtual machine (VM) is known in the art as an abstraction, e.g., a software emulation, of an actual physical machine/computer system, the VM capable of running an operating system and
30 other software. In some embodiments, hypervisor **50** includes software configured to create or

enable a plurality of virtualized devices, such as a virtual processor and a virtual controller hub, and to present such virtualized devices to software in place of the real, physical devices of host system **10**. Such operations of hypervisor **50** are commonly known in the art as exposing a virtual machine. In some embodiments, hypervisor **50** allows a multiplexing (sharing) by multiple virtual machines of hardware resources of host system **10**. Hypervisor **50** may further manage such multiplexing so that each guest VM **52a-b** operates independently and is unaware of other VMs executing concurrently executing on host system **10**. Examples of popular hypervisors include the VMware vSphere™ from VMware Inc. and the open-source Xen hypervisor, among others.

[0029] Each VM **52a-b** may execute a guest operating system (OS) **54a-b**, respectively. A set of exemplary applications **56a-d** generically represent any software application, such as word processing, image processing, media player, database, calendar, personal contact management, browser, gaming, voice communication, data communication, and anti-malware applications, among others. Operating systems **54a-b** may comprise any widely available operating system such as Microsoft Windows®, MacOS®, Linux®, iOS®, or Android™, among others. Each OS **54a-b** provides an interface between applications executing within the respective VM and the virtualized hardware devices of the respective VM. In the following description, software executing on a virtual processor of a virtual machine is said to execute within the respective virtual machine. For instance, in the example of Fig. 2-A, applications **56a-b** are said to execute within guest VM **52a**, while applications **56c-d** are said to execute within guest VM **52b**. In contrast, hypervisor **50** is said to execute outside, or below, guest VMs **52a-b**.

[0030] Fig. 3 shows an exemplary configuration of a virtual machine **52**, as exposed by hypervisor **50**. VM **52** may represent any of VMs **52a-b** of Fig. 2-A. VM **52** includes a virtualized processor **112**, a virtualized memory unit **114**, virtualized input devices **116**, virtualized output devices **118**, virtualized storage **120**, virtualized network adapters **122**, and a virtualized controller hub **124**. Virtualized processor **112** comprises an emulation of at least some of the functionality of processor **12**, and is configured to receive for execution processor instructions forming part of software such as an operating system and other applications. Software using processor **112** for execution is deemed to execute within virtual machine **52**. In some embodiments, virtualized memory unit **114** comprises addressable spaces for storing and

retrieving data used by virtualized processor **112**. Other virtualized devices (e.g., virtualized input, output, storage, etc.) emulate at least some of the functionality of the respective physical devices of host system **10**. Virtualized processor **112** may be configured to interact with such virtualized devices as it would with the corresponding physical devices. For instance, software
5 executing within VM **52** may send and/or receive network traffic via virtualized network adapter(s) **122**. In some embodiments, hypervisor **50** may expose only a subset of virtualized devices to VM **52** (for instance, only virtualized processor **112**, virtualized memory **114**, and parts of hub **124**). Hypervisor **50** may also give a selected VM exclusive use of some hardware devices of host system **10**. In one such example, VM **52a** (Fig. 2-A) may have exclusive use of
10 input devices **16** and output devices **18**, but lack a virtualized network adapter. Meanwhile, VM **52b** may have exclusive use of network adapter(s) **22**. Such configurations may be implemented, for instance, using VT-d® technology from Intel®.

[0031] Modern processors implement a hierarchy of processor privilege levels, also known in the art as protection rings. Each such ring or level is characterized by a set of actions and/or
15 processor instructions that software executing within the respective ring is allowed to carry out. Exemplary privilege levels/rings include user mode (ring 3) and kernel mode (ring 0). Some host systems configured to support hardware virtualization may include an additional ring with the highest processor privileges (e.g., ring -1, root mode, or VMXroot on Intel® platforms). In some embodiments, hypervisor **50** takes control of processor **12** at the most privileged level (ring
20 -1), thus creating a hardware virtualization platform exposed as a virtual machine to other software executing on host system **10**. An operating system, such as guest OS **54a** in Fig. 2-A, executes within the virtual environment of the respective VM, typically with lesser processor privilege than hypervisor **50** (e.g., in ring 0 or kernel mode). Common user applications, such as **56a-b**, typically execute at lesser processor privilege than OS **34a** (e.g., in ring 3 or user mode).
25 Some parts of applications **56a-b** may execute at kernel privilege level, while some parts of OS **34a** may execute in user mode (ring 3). When a software object attempts to execute an action or instruction requiring processor privileges higher than allowed by its assigned protection ring, the attempt typically generates a processor event, such as an exception or a fault, which transfers control of processor **12** to an entity (e.g., handler routine) executing in a ring with
30 enough privileges to carry out the respective action.

[0032] In particular, an attempt to perform certain actions or to execute certain instructions from within a guest VM may trigger a special category of processor events, herein generically termed VM suspend events. In some embodiments, a VM suspend event suspends execution of the current thread within a guest VM and switches processor 12 to executing a handler routine.

5 Exemplary VM suspend events include, among others, a VM exit event (e.g., VMExit on Intel® platforms) and a virtualization exception (e.g. #VE on Intel® platforms). VM exit events switch processor 12 to executing a handler routine outside the respective guest VM, typically at the level of hypervisor 50. Virtualization exception may switch processor 12 to executing a handler routine within the respective guest VM, instead of exiting the respective VM.

10 [0033] Exemplary instructions triggering a VM suspend event include VMCALL on Intel® platforms. VM suspend events may also be triggered by other events, such as memory access violations. In one such example, when a software object executing within a VM attempts to write to a section of memory marked as non-writable, or to execute code from a section of memory marked as non-executable, processor 12 may generate a VM exit event. Such VM-

15 switching mechanisms allow, for example, a computer security program to protect a virtual machine from outside the respective VM. The computer security program may intercept VM exit events occurring in response to certain actions performed by software running inside the VM, actions which may be indicative of a security threat. The computer security program may then block and/or further analyze such actions, potentially without the knowledge of in-VM

20 software. Such configurations may substantially strengthen computer security.

[0034] In some embodiments (e.g., Fig. 2-A), hypervisor 50 includes a computer security module (CSM) 60, configured to perform such computer security operations, among others. Module 60 may be incorporated into hypervisor 50 (for instance as a library), or may be delivered as a computer program distinct and independent from hypervisor 50, but executing at

25 the privilege level of hypervisor 50. A single module 60 may be configured to protect multiple guest VMs executing on host system 10. Security operations carried out by module 60 may include detecting an action performed by a process executing within a guest VM (e.g., calling certain functions of the OS, accessing a registry of the OS, downloading a file from a remote location, writing data to a file, etc.). Other security operations of module 60 may comprise

30 determining an address of a memory section containing a part of a software object executing

within a guest VM, accessing the respective memory section, and analyzing a content stored within the respective memory section. Other examples of security operations include intercepting and/or restricting access to such memory sections, e.g., preventing the over-writing of code or data belonging to a protected process, and preventing the execution of code stored in certain memory pages. In some embodiments, CSM 60 includes a VM exit event handler 61
5 configured to intercept VM exit events occurring within guest VMs 52a-b. In an alternative embodiment, handler 61 may be a distinct module (e.g., a library) of hypervisor 50, separate from CSM 60, which intercepts VM exit events and selectively transfers control to CSM 60 after determining a reason and/or a type of each VM exit that occurred.

10 [0035] Fig. 2-B illustrates an alternative embodiment wherein computer security module 60 protects a guest VM 52c from outside the respective VM. In such embodiments, processor 12 may be configured to generate a virtualization exception (instead of a VM exit event, as described above in relation to Fig. 2-A) when a memory access violation occurs. In the exemplary embodiment of Fig. 2-B, a virtualization exception handler 63 executes within
15 VM 52c, for instance at the privilege level of an operating system 54c, and is configured to intercept virtualization exceptions and interface with CSM 60.

[0036] Communication between handler 63 and CSM 60 may proceed according to any inter-process communication method known in the art. To transmit data from within the protected VM to the level of hypervisor 50, some embodiments of handler 63 use a specialized instruction
20 (e.g., VMCALL on Intel® platforms) to transfer control of processor 12 from the respective VM to hypervisor 50. The data being transmitted may be placed by exception handler 63 in a predetermined section of memory shared with CSM 60. To transmit data to handler 63, some embodiments of CSM 60 may inject an interrupt into VM 52c, the interrupt handled by handler 63. The respective data may be transferred again through the shared memory section
25 described above.

[0037] In yet another embodiment, illustrated in Fig. 2-C, both CSM 60 and handler 63 execute within the protected VM, for instance in kernel mode (ring 0). Such embodiments may also employ virtualization exceptions to detect memory access violations. Deciding between configurations 2-A-B-C may comprise evaluating a trade-off between performance and security.

VM exit events are relatively costly in terms of computation, typically requiring loading and/or unloading of large data structures into/from memory with each exit and re-entry cycle. Hence, configurations such as **2-A** may require more computation to intercept an event than configurations such as **2-B-C**. On the other hand, keeping critical security components such as CSM **60** and handlers **61-63** outside the protected VM (as in examples **2-A-B**) may strengthen security, since it may be more difficult for malware executing within the respective VM to interfere with the operation of such components.

[0038] To be able to protect a guest VM in a configuration as illustrated in Figs. **2-A-B** (i.e., from outside the respective VM), some embodiments of CSM **60** employ address translation data structures and/or address translation mechanisms of processor **12**. Virtual machines typically operate with a virtualized physical memory (see, e.g., memory **114** in Fig. **3**), also known in the art as guest-physical memory. Virtualized physical memory comprises an abstract representation of the actual physical memory **14**, for instance as a contiguous space of addresses, commonly termed guest-physical addresses (GPA). Each such address space is uniquely attached to a guest VM, with parts of said address space mapped to sections of physical memory **14** and/or physical storage devices **20**. In systems configured to support virtualization, such mapping is typically achieved using hardware-accelerated, dedicated data structures and mechanisms controlled by processor **12**, known as second level address translation (SLAT). Popular SLAT implementations include extended page tables (EPT) on Intel® platforms, and rapid virtualization indexing (RVI)/nested page tables (NPT) on AMD® platforms. In such systems, virtualized physical memory may be partitioned in units known in the art as pages, a page representing the smallest unit of virtualized physical memory individually mapped to physical memory via mechanisms such as EPT/NPT, i.e., mapping between physical and virtualized physical memory is performed with page granularity. All pages typically have a predetermined size, e.g., 4 kilobytes, 2 megabytes, etc. The partitioning of virtualized physical memory into pages is usually configured by hypervisor **50**. In some embodiments, hypervisor **50** also configures the SLAT structures, and therefore configures address translation between physical memory and virtualized physical memory. Such address translations are known in the art as guest-physical to host-physical (GPA-to-HPA) translations.

[0039] In some embodiments, the operating system executing within a VM sets up a virtual memory space for each process executing within the respective VM, said virtual memory space representing an abstraction of physical memory. Process virtual memory typically comprises a contiguous space of addresses, commonly known in the art as guest-virtual addresses (GVA) or guest-linear addresses (GLA). In some embodiments, process virtual memory spaces are also partitioned into pages, such pages representing the smallest unit of virtual memory individually mapped by the OS to the virtualized physical memory of the respective VM, i.e., virtual to virtualized-physical memory mapping is performed with page granularity. The OS may configure a dedicated data structure, such as a page table, used by the virtualized processor of the respective VM to perform guest virtual to guest physical, or GVA-to-GPA address translations.

[0040] Fig. 4 illustrates an exemplary memory address translation in the embodiment of Fig. 2-A. Following exposure by hypervisor 50, guest VM 52a sees a virtualized physical memory space 114a as its own physical memory space. A process executing within guest VM 52a is assigned a virtual memory space 214a by guest OS 54a. When the process attempts to access memory at a guest-virtual address 62, GVA 62 is translated by the (virtualized) MMU of guest VM 52a into a guest-physical address 64 within virtualized physical memory space 114a. GVA-to-GPA translation 70a may proceed, for instance, according to page tables configured and controlled by guest OS 34a. GPA 64 is further mapped by the MMU to a host-physical address (HPA) 66 within physical memory 14 of host system 10. GPA-to-HPA translation 70b may proceed, for instance, according to SLAT structures configured by hypervisor 50.

[0041] Each process executing below guest VMs 52a-b is typically assigned a virtual memory space addressable via what is known in the art as host-virtual addresses (HVA). In the example of Fig. 4, hypervisor 50 sets up a virtual memory space 214b for computer security module 60. CSM 60 may then reference HPA 66 via a HVA 68. When module 60 is integrated within hypervisor 50, for instance as a library, memory space 214b may coincide with the virtual memory space of hypervisor 50. To manage such spaces, hypervisor 50 may configure dedicated data structures and mechanisms (e.g. page tables) used by the MMU to perform HVA-to-HPA translations such as translation 70c.

[0042] In some embodiments, hypervisor **50** and/or CSM **60** may set access permissions for each of a subset of physical memory pages. Such memory pages may be used, for instance, by certain critical guest processes executing within a protected VM, such as processes of the OS and/or anti-malware routines. Access permissions indicate, for instance, whether the respective page may be read from and written to, and whether software is allowed to execute code from the respective page. Access permissions may be indicated, for instance, as a part of the SLAT entry representing the respective memory page. Some host systems may allow setting access permissions with sub-page granularity.

[0043] Hypervisor **50** and/or CSM **60** may further configure processor **12** to generate a VM suspend event when software executing within a guest VM attempts to access memory in a manner that violates access permissions (e.g., to write to a memory page marked as non-writable). Such an attempt is hereby termed memory access violation. The respective VM suspend event may be a VM exit event in configurations such as Fig. **2-A**, and a virtualization exception in configurations such as Fig. **2-B-C**.

[0044] Some embodiments of the present invention introduce changes to the structure and functionality of a conventional hardware processor, to enable the processor to function more efficiently in hardware virtualization configurations. Fig. **5** shows exemplary hardware components of processor **12** according to some embodiments of the present invention. The illustrated components are meant as generic devices performing the described functionality; structural details may vary substantially among implementations. For instance, each illustrated component may comprise multiple interconnected subsystems, not necessarily in physical proximity to each other. The illustrated components are not exhaustive; processor **12** may include many other components (e.g., scheduler, interrupt controller, various caches, etc.), which were omitted from Fig. **5** to simplify presentation.

[0045] Processor **12** may include logic/circuitry configured to carry out various stages of a processor pipeline. For instance, an instruction decoder **30** performs instruction decoding operations, which may include translating each processor instruction into a set of elementary processor operations and/or micro-ops. A set of execution units **36** connected to decoder **30** may

perform the execution stage of the pipeline. Exemplary execution unit(s) **36** include, among others, an arithmetic logic unit (ALU) and a floating-point unit (FPU).

[0046] In some embodiments, the execution stage of the pipeline for an instruction comprises determining a result of applying an operator of the respective instruction to an operand of the respective instruction. Such results may comprise, among others, a memory address, a value to be committed to a memory address or to a processor register (e.g., to a general purpose register such as AX, a model-specific register – MSR, a control register such as EFLAGS, or a hidden register such as the hidden part of an x86 segment register, also known as a descriptor cache), a value of the instruction pointer (e.g., RIP), and a value of the stack pointer (e.g., RSP).

[0047] The operand(s) of an instruction may be explicit in the statement of the instruction, or may be implicit. An exemplary x86 instruction with implicit operands is the STC instruction, which sets the carry flag of the EFLAGS control register of the processor to 1. In some embodiments, the register EFLAGS and the value 1 are interpreted as (implicit) operands, although they do not appear explicitly in the statement of the STC instruction.

[0048] A commit unit **38** may perform the commit stage of the pipeline, i.e., to store the output of execution unit(s) **36** in memory **14** and/or to update the contents of certain processor registers to reflect changes/results produced by the execution stage. Commit unit **38** may comprise logic modules known in the art as retirement units.

[0049] A memory access module **34** connected to decoder **30** and execution unit(s) **36** includes logic configured to interface with memory **14**, e.g., to fetch instructions from memory, to load data from memory, and to store results of execution of processor instructions to memory. In some embodiments, memory access module comprises an MMU configured to perform the virtual-to-physical address translations necessary for memory access.

[0050] Modern processors typically support out-of-order and/or speculative execution of processor instructions. In such systems, multiple instructions are concurrently fetched, decoded, and executed by the same execution unit(s) **36**. Results of such executions are then committed in-order, to preserve the intended flow of the respective computer program. Such configurations are used, for instance, in conjunction with branch prediction algorithms, to enhance the

performance of processor **12**. In some embodiments configured for out-of-order execution, processor **12** may further comprise a dispatcher unit **32** coupled to decoder **30** and to execution units **36**, and a register file **40** coupled to execution unit(s) **36** and commit unit **38**. Dispatcher unit **36** may schedule individual micro-ops for execution, and maintain a mapping associating
5 each micro-op with its respective instruction, to control the order of execution and commit. Register file **40** comprises an array of internal processor registers, organized, for instance, as a reorder buffer. Register file **40** may further comprise logic enabling dispatcher unit **36** to associate a row of registers of file **40** to each scheduled micro-op, an operation known in the art as register renaming. In such configurations, each such row of registers may hold, for instance,
10 the values of the general purpose and/or status registers of processor **12**, said values corresponding to an intermediate stage of execution of a certain processor instruction.

[0051] Processor **12** may further include a virtual machine control unit **38** configured to manage virtual machine state data. In some embodiments, a virtual machine state object (VMSO) comprises a data structure used internally by processor **12** to represent the current state of each
15 virtualized processor exposed on host system **10**. Exemplary VMSOs include the virtual machine control structure (VMCS) on Intel® platforms, and the virtual machine control block (VMCB) on AMD® platforms. VMSOs are typically set up by hypervisor **50** as part of exposing each virtual machine. In some embodiments, processor **12** associates a region in memory with each VMSO, so that software may reference a specific VMSO using a memory
20 address or pointer (e.g., VMCS pointer on Intel® platforms).

[0052] Each VMSO may comprise a guest state area and a host state area, the guest state area holding the CPU state of the respective guest VM, and the host state area storing the current state of hypervisor **50**. In some embodiments, the guest-state area of the VMSO includes contents of the control registers (e.g., CR0, CR3, etc.), instruction pointer (e.g., RIP), general-purpose
25 registers (e.g., EAX, ECX, etc.), and status registers (e.g., EFLAGS) of the virtual processor of the respective guest VM, among others. The host state area of the VMSO may include a pointer (e.g., an EPT pointer on Intel® platforms) to a SLAT data structure configured for GPA-to-HPA address translations for the respective guest VM.

[0053] In some embodiments, processor 12 may store a part of a VMSO within dedicated internal registers/caches, while other parts of the respective VMSO may reside in memory. At any given time, at most one VMSO (herein termed the current VMSO) may be loaded onto the processor, identifying the virtual machine currently having control of processor 12. Modern processors are typically configured for multithreading. In such configurations, physical processor 12 may operate a plurality of cores, each core further comprising multiple logical processors, wherein each logical processor may process an execution thread independently of, and concurrently with, other logical processors. Multiple logical processors may share some hardware resources, for instance, a common MMU. In a multithreaded embodiment, a distinct VMSO may be loaded onto each distinct logical processor.

[0054] When processor 12 switches from executing the respective VM to executing hypervisor 50 (e.g., upon a VM exit), processor 12 may save the state of the respective VM to the guest state area of the current VMSO. When processor 12 switches from executing a first VM to executing a second VM, the VMSO associated to the first VM is unloaded, and the VMSO associated to the second VM is loaded onto the processor, the second VMSO becoming the current VMSO. In some embodiments, such loading/unloading of VMSO data to/from processor 12 is performed by virtual machine control module 38. Module 38 may further carry out the retrieval and/or saving of VMSO data from/to memory 14.

[0055] In some embodiments, processor 12 further comprises a suspend event register 44 connected to execution unit(s) 36 and/or to commit unit 38, and configured to store instruction-specific data associated with a guest instruction, wherein execution of said guest instructions has caused a VM suspend event (e.g., a VM exit or a virtualization exception). In some embodiments, suspend event register 44 is an exposed register, accessible to software executing on host system 10, i.e., data stored in register 44 may be readable by software such as security module 60. In one such example, suspend event register 44 includes a model-specific register (MSR) of processor 12. Some embodiments may restrict access to register 44 to a subset of software objects, selected according to a criterion such as processor privilege (e.g., only ring -1 or root mode) or object type (e.g., only drivers). Some embodiments may restrict software access to register 44 only to a subset of operations (e.g., read only).

[0056] Fig. 6 shows an exemplary set of fields of suspend event register 44 according to some embodiments of the present invention. Register 44 may include a disassembly field 46a and an execution result field 46b. Disassembly field 46a may store data resulting from disassembling the respective guest instruction.

5 [0057] Fig. 7 shows an assembly language representation 45 of an exemplary Intel® x86 processor instruction. The illustrated instruction instructs the processor to increment the content stored in memory at the (virtual) address $EBX + 4 * ECX + 0x02$, by the value currently stored in register AX. The respective instruction is represented in memory as a machine code 47; the translation between representations 45 and 47 is typically done by a compiler or assembler.

10 Machine code representation 47 has a generic form 48, comprising a sequence of encoding fields 49a-f, which are specific to each ISA and/or to each family of processors. On Intel® processors, such encoding fields include Prefix, Opcode, Mod R/M, SIB, Displacement, and Immediate fields. Fig. 7 shows the instance of each encoding field for the given exemplary x86 instruction. In some ISAs, machine code representation 47 may have variable length, for instance, some encoding fields may appear in the machine code representation of certain instructions, but may not appear in the representation of other instructions. In the example of Fig. 7, representation 47 lacks an Immediate encoding field.

20 [0058] In some embodiments, disassembling an instruction comprises parsing the machine code representation of the instruction to identify individual instruction encoding fields, such as fields 49a-f in Fig. 7. Disassembling may further comprise extracting the content of individual encoding fields, and/or using such content to determine a set of semantic elements of the guest instruction. Such semantic elements may include an operator (e.g., MOV, ADD, etc.) and an operand (e.g., AX, $[EBX + 4 * ECX + 0x02]$) of the instruction, among others. Instruction disassembly may be carried out, at least in part, by instruction decoder 30 and/or by execution unit(s) 36.

25 [0059] In the example of Fig. 7, disassembling the illustrated instruction may include identifying the ADD operator and/or determining according to machine code 47 that the respective instruction has two operands, that one of the operands is the content of the AX register, that the second operand is a content of memory, and determining an expression (e.g.,

EBX+4*ECX+0x02) of the respective memory address. In some embodiments, disassembling the instruction further comprises computing a memory address indicated by an operand of the respective instruction (e.g., the value of the expression EBX+4*ECX+0x02). In another example, wherein the guest instruction is a relative jump instruction (e.g., JMP \$+10 on x86 platforms, represented in machine code as 0xEB 0x08), disassembling the guest instruction may
5 comprise calculating an absolute memory address of the destination according to the address of the guest instruction, to the length of the guest instruction, and/or to the size of the relative jump.

[0060] In some embodiments, disassembly field 46a of register 44 (Fig. 6) is configured to store a content of individual instruction encoding fields of the guest instruction, such as the Opcode, Mod R/M, and SIB fields. Other exemplary content of field 46a includes an operator identifier
10 indicating the operator of the respective instruction, and an indicator of an operand of the respective instruction. The operand indicator may further include an identifier of a processor register (e.g., AX), and a flag indicating, for instance, whether the respective operand is the content of a register or a content of memory. Disassembly field 46a may further comprise a
15 memory address (e.g., GVA, GPA, and/or HPA) indicated by an operand. The structure of disassemble field 46a may be ISA and/or platform-specific.

[0061] In some embodiments, execution result field 46b of suspend event register 44 may store data indicative of a result of executing the respective processor instruction. Such results may include a value of a status register (e.g., FLAGS), a value of an instruction pointer (e.g., RIP),
20 and a value of a general purpose register (e.g., EAX) resulting from executing the respective instruction. Field 46b may further comprise a value to be committed to memory as a result of executing the respective instruction, a size of the respective value (e.g., byte, word, etc.) and/or a memory address where the respective value is to be committed.

[0062] In some embodiments, execution unit(s) 36 and/or commit unit 38 may be configured to
25 determine whether execution of a guest instruction causes a VM processor event (such as a VM exit or virtualization exception), and when yes, to save instruction disassembly data to suspend event register 44 before generating the respective event. Processor 12 may be further configured to delay the generation of the processor event until completion of the execution stage of the respective guest instruction, and to save a result of executing the respective instruction to event

register **44** instead of committing such results to memory and/or to a general purpose register of processor **12**. To avoid committing results of such instructions, processor **12** may be configured to generate the VM processor event before the commit stage of the pipeline for the respective instruction. Such functionality will be further detailed below.

5 [0063] Fig. 8 shows a detailed, exemplary sequence of steps performed by processor **12** to execute a guest instruction according to some embodiments of the present invention. Fig. 8 shows an embodiment, wherein processor **12** is configured to generate a VM exit event in response to a memory access violation. A skilled artisan will appreciate that the present description may easily be modified to cover an embodiment, which generates other VM suspend
10 events (such as a virtualization exception) instead of a VM exit event. "Guest instruction" is a term used herein to denote a processor instructions forming part of a computer program executing within a guest VM, such as VMs **52a-b** in Fig. 2-A.

[0064] A step **302** attempts to fetch the guest instruction. When the fetch attempt fails, a step **303** may determine whether the failure is caused by a memory access violation (for instance,
15 when the guest instruction resides in a memory page marked as non-executable in a SLAT structure of the guest VM). When no, in a step **306**, processor **12** generates a VM exit event and transfers execution to an event handler, such as handler **61** in Fig. 2-A. When failure to fetch the guest instruction is caused by a memory access violation, such a failure may be indicative of a security program (e.g., anti-malware module) trying to protect a content of the respective
20 memory page. One exemplary memory section typically protected from execution in this manner stores an execution stack of a guest process. Marking the stack as non-executable may protect the guest process, for instance, from a stack exploit. In such situations, some embodiments may re-attempt to fetch the guest instruction, ignoring the respective memory access permissions (step **305**). In a step **307**, the fetched guest instruction is marked with a
25 dedicated flag, to indicate that the respective instruction was "force-fetched", i.e., was fetched while breaking memory access permissions. Processor **12** may then proceed to a step **308**.

[0065] Following the fetch stage, step **308** decodes and dispatches the guest instruction. In a step **310**, the guest instruction is launched into execution. When executing the guest instruction satisfies a criterion for VM exit, wherein the criterion is not related to memory access,

processor **12** proceeds to a step **322** detailed below. Such VM exits may be triggered in a variety of situations. For instance, the guest instruction may be a specialized instruction, such as VMCALL, which automatically trigger a VM exit event when called from within a guest VM. Another exemplary reason for VM exit, which is not related to memory access, is the occurrence of a hardware event (e.g., an interrupt) during execution of the guest instruction.

[0066] When executing the guest instruction causes a memory access violation (for instance, when the guest instruction instructs the processor to write a result to a memory page marked as non-writable), a conventional processor typically suspends execution of the guest instruction, flushes the processor pipeline(s) and generates a VM suspend event (e.g. VMExit). In contrast, in some embodiments of the present invention, execution of the guest instruction is not suspended. Instead, in a step **318**, the VM exit event is delayed until the execution stage of the pipeline for the guest instruction finishes. However, in some embodiments, the results of the completed execution stage are not committed, as would happen in conventional systems. Instead, in a step **320**, processor **12** may instruct commit unit **38** to store the results of the completed execution stage of the guest instruction in suspend event register **44**. Such functionality may be achieved, for instance, using an activation signal to switch commit unit **38** from committing results to memory and/or general purpose registers of processor **12**, to storing results in register **44** when a memory access violation has occurred. The control signal may indicate whether execution of the guest instruction has caused a memory access violation. Commit unit **38** may receive such a signal, for instance, from the MMU via memory access module **34**. In some embodiments, step **320** comprises commit unit **38** retrieving a result of executing the guest instruction from register file **40**.

[0067] In an alternative embodiment, instead of saving execution results of the guest instruction to register **44**, step **320** may save such results to a dedicated memory region, such as the guest state area of the VMSO of the respective guest VM. In yet another embodiment, processor **12** may transmit such results to VM exit handler **61** upon executing the VM exit (step **306**).

[0068] In a step **322**, processor **12** may store data resulting from disassembling the guest instruction to suspend event register **44** (and/or to memory as described above). In some embodiments, step **322** includes writing a content of an individual encoding field of the guest

instruction to register **44**. Instruction disassembly data may be produced by instruction decoder **30** and/or execution unit(s) **36** in the process of decoding and/or executing the guest instruction; step **322** may include retrieving such data from the respective processor module. After storing execution results and/or disassembly data for the guest instruction, processor **12** may generate a VM exit event (step **306**).

[0069] When execution of the current guest instruction proceeds without causing memory access violations (step **314**) and without non-memory related reasons for a VM exit (step **312**), a step **315** may determine whether the current guest instruction was force-fetched (see steps **305-307** above). When no, a step **316** commits results of the execution to memory and/or to general purpose processor registers. When the current guest instruction is force-fetched, some embodiments may treat the respective instruction as an instruction causing a memory access violation, i.e., by waiting for the respective instruction to complete the execution stage of the pipeline, storing results and/or instruction disassembly data to register **44**, before generating a VM exit event (see steps **318-320-322-306** above).

[0070] Fig. **9** shows an exemplary sequence of steps performed by a guest VM and/or by computer security module **60** (Figs. **2-A-B**) according to some embodiments of the present invention related to computer security. A guest process, such as an application (e.g., **56a** in Fig. **2-A**) or a process of the operating system (e.g., guest OS **54a** in Fig. **2-A**) may execute within the guest VM, advancing stepwise through a sequence of guest instructions (step **332**). Execution of the guest process continues until a VM exit is generated, according, for instance, to a scenario described above in relation to Fig. **8**. A skilled artisan may appreciate how the description may be adapted to a system wherein processor **12** generates a virtualization exception instead of a VM exit event, and wherein an exception handler executing within the guest VM (e.g., handler **63** in Fig. **2-B**) is configured to intercept the respective exception.

[0071] In a step **336**, handler **61** intercepts the VM exit event, which is analyzed for evidence of a security threat. When the event indicates a security threat (e.g., an operation executed with malicious intent), in a step **340**, CSM **60** may take protective action against the guest process and/or against the guest VM. Such action may include, among others, blocking the execution of

the guest process, returning an error message or a set of dummy results to the guest process, and alerting an administrator of the host system.

[0072] When the VM exit event is not indicative of a security threat, a step 342 determines whether the results of executing the guest instruction are available (either in event register 44 of processor 12 or in memory). When no, CSM 60 advances to a step 348 detailed below. When yes, a step 344 retrieves the respective results from register 44 and/or memory (e.g., guest state area of the VMSO of the respective guest VM). In a step 346, CSM 60 may apply the results of executing the current guest instruction. In some embodiments, step 346 comprises a set of operations carried out in conventional systems at the commit stage. For instance, step 346 may include updating values of general purpose, control, and status processor registers of the virtualized processor of the respective guest VM. In some embodiments, such registers are accessible within the guest state area of the VMSO of the respective guest VM. Step 346 may further include saving some results to memory addresses indicated by an operand of the current guest instruction. Step 346 may further include incrementing the instruction pointer (e.g., RIP in x86 platforms), to show that execution of the current guest instruction is complete.

[0073] Some embodiments of the present invention add a dedicated instruction to the current instruction set architecture (ISA) of processor 12, the new instruction instructing processor 12 to apply a result of execution of a guest instruction directly, from below the guest VM executing the respective guest instruction. The new instruction (an exemplary mnemonic is VMAPPLY) may carry out operations of step 346 (Fig. 9), e.g., copy contents from suspend event register 44 to virtual registers of the virtualized processor of the respective guest VM and/or to memory.

[0074] In some embodiments, step 346 may further verify whether the current guest instruction is an atomic instruction (e.g., as indicated by a LOCK prefix). When yes, instead of applying results directly to registers of the guest and/or to memory, step 346 may force a re-execution of the current guest instruction upon returning to guest VM (see step 356 below).

[0075] When execution results of the current guest instruction are not available (for instance, when the current VM exit was caused by a privileged instruction such as VMCALL), in a step 348, computer security module 60 determines whether disassembly data is available for the current guest instruction. When yes, in a step 350, CSM 60 may retrieve such data, for instance

from disassembly field **46a** of register **44** (see e.g., Fig. **6**). CSM **60** may then proceed to emulate the current guest instruction according to the retrieved disassembly data (step **354**).

[0076] When no disassembly data is available, a step **352** may disassemble the current guest instruction before proceeding with emulation. In a step **356**, CSM **60** may re-launch the
5 respective guest VM (e.g., by issuing a VMRESUME instruction on Intel® platforms). In some embodiments wherein step **346** includes a modification of the instruction pointer, execution of the guest process will start with the processor instruction immediately following the current guest instruction, or with a processor instruction indicated by the current guest instruction (e.g., in the case of control flow-changing instructions such as JMP, CALL, etc.).

10 [0077] The exemplary systems and methods described above allow a host system, such as a computer or a smartphone, to efficiently carry out computer security tasks when operating in a hardware virtualization configuration. Security tasks may include, among others, protecting the host system against malware such as computer viruses and spyware. In some embodiments, the host system is configured to execute an operating system and a set of software applications
15 within a virtual machine. A security module may execute outside the respective virtual machine, for instance at the level of a hypervisor, and may protect the respective virtual machine against malware.

[0078] In some embodiments, the security module identifies a section of memory (e.g. a set of memory pages) containing code and/or data which is critical to the security of the protected VM,
20 and configures access permissions for the respective section of memory. Such access permissions may indicate, for instance, that the respective section of memory is non-writable and/or non-executable. The security module may further configure the processor of the host system to generate a VM suspend event (such as a VM exit or a virtualization exception) in response to a memory access violation, e.g., when software executing within the protected VM
25 attempts to write to a section of memory marked as non-writable, or to execute code from a section of memory marked as non-executable. The security module may then intercept such processor events via an event handler, and may determine whether such events are indicative of a computer security threat. In configurations wherein the security module executes outside the

protected VM, the activity of the security module is potentially invisible to software executing within the protected VM, including malware.

[0079] In conventional systems, intercepting VM suspend events proceeds according to methods generically known in the art as “trap-and-emulate”. In one example of a conventional technique, after determining which instruction caused the respective event (e.g., a VM exit), the anti-malware program emulates the respective instruction before returning execution to the protected VM, and modifies the instruction pointer to indicate that the respective instruction has already been executed. Without the emulation step, returning execution to the protected VM would typically re-trigger the VM exit, thus creating an infinite loop.

[0080] Conventional trap-and-emulate systems and methods may therefore require the anti-malware program to include an instruction disassembler and/or an instruction emulator. Such components may be complex to develop and maintain and may not be portable, for instance, from one processor to another. Moreover, in conventional systems, the disassembly and/or emulation steps are typically carried out for every VM suspend event, placing a considerable computational burden onto the host system. In contrast, some embodiments of the present invention eliminate the need for a disassembler and/or an emulator, substantially accelerating computer security operations.

[0081] Some embodiments of the present invention introduce changes to the configuration and operation of conventional processors, enabling such processors to operate more efficiently in hardware virtualization configurations.

[0082] In some embodiments, the processor is configured to, in response to determining that a condition for generating a VM suspend event is satisfied, to save disassembly data determined for the currently executing guest instruction to a predetermined location, such as a specific processor register or a specific memory area (e.g., the guest state area of a virtual machine state object used by the hypervisor to manage execution of the protected VM). Such disassembly data may include, for instance, indicators of an operator and an operand of the current guest instruction. In one exemplary embodiment, disassembly data comprises contents of individual encoding fields of the guest instruction, such as the opcode, mod R/M, and SIB fields, among others.

[0083] Such improvements may be especially beneficial for computer security applications, facilitating an efficient protection of a virtual machine from outside, e.g., from the level of a hypervisor exposing the respective VM. When compared to a conventional computer security solution, some embodiments of the present invention allow a substantial reduction in computation, by eliminating the instruction disassembly stage from the operation of security software configured to intercept and analyze VM exit events.

[0084] It will be clear to a skilled artisan that the above embodiments may be altered in many ways without departing from the scope of the invention. Accordingly, the scope of the invention should be determined by the following claims and their legal equivalents.

CLAIMS

What is claimed is:

1. A host system comprising at least one hardware processor configured to execute a virtual machine and a computer security program, wherein the at least one processor is further configured to:
 - determine whether executing a guest instruction within the virtual machine causes a violation of a memory access permission; and
 - in response, when executing the guest instruction causes the violation:
 - write a part of a machine code representation of the guest instruction to a predetermined location accessible to the computer security program;
 - suspend the execution of the guest instruction; and
 - in response to suspending the execution of the guest instruction, switch to executing the computer security program, wherein the computer security program is configured to determine whether the violation is indicative of a computer security threat.
2. The host system of claim 1, wherein the machine code representation comprises a sequence of encoding fields, and wherein the part comprises a content of an individual encoding field of the guest instruction.
3. The host system of claim 2, wherein the individual encoding field is selected from a group consisting of a prefix field, an opcode field, a mod R/M field, a SIB field, a displacement field, and an immediate field of the machine code representation.
4. The host system of claim 1, wherein the part is indicative of an operator of the guest instruction.

- 1 5. The host system of claim 1, wherein the part is indicative of an operand of the guest
2 instruction.
3
- 1 6. The host system of claim 1, wherein the at least one hardware processor is further
2 configured to, in response to determining whether executing the guest instruction
3 causes the violation, when executing the guest instruction causes the violation, write a
4 memory address indicated by the guest instruction to the predetermined location.
5
- 1 7. The host system of claim 6, wherein the memory address is a guest-physical
2 address (GPA) located within a memory space of the virtual machine, the
3 GPA determined by translating a virtual address indicated by the machine
4 code representation.
5
- 1 8. The host system of claim 6, wherein the memory address is a host-physical
2 address (HPA) located within a physical memory of the host system, the HPA
3 determined by translating a virtual address indicated by the machine code
4 representation.
5
- 1 9. The host system of claim 6, wherein the memory address is determined
2 according to a memory location of the guest instruction.
3
- 1 10. The host system of claim 1, wherein the at least one hardware processor is further
2 configured to, in response to determining whether executing the guest instruction
3 causes the violation, when executing the guest instruction causes the violation, write a
4 indicator of a processor register to the predetermined location, the register indicated
5 by the guest instruction.
6
- 1 11. The host system of claim 1, wherein the computer security program executes within
2 the virtual machine.
3

12. The host system of claim 1, wherein the computer security program executes outside the virtual machine.

13. The host system of claim 1, wherein the predetermined location comprises a predetermined register of the at least one hardware processor.

14. The host system of claim 13, wherein the predetermined register is a model-specific register (MSR) of the at least one hardware processor.

15. The host system of claim 1, wherein the predetermined location comprises a predetermined section of a memory of the host system.

16. The host system of claim 1, wherein the predetermined location comprises a data structure indicative of a current state of the virtual machine.

17. A method of protecting a host system from computer security threats, the method comprising:
in response to receiving a guest instruction for execution, employing at least one hardware processor of the host system to determine whether executing the guest instruction causes a violation of a memory access permission, wherein the guest instruction executes within a virtual machine exposed by the host system; and
in response to determining whether the guest instruction causes the violation, when executing the guest instruction causes the violation:
employing the at least one hardware processor to write a part of a machine code representation of the guest instruction to a predetermined location accessible to the computer security program;
employing the at least one hardware processor to suspend the execution of the guest instruction; and
in response to suspending the execution of the guest instruction, switching to executing a computer security program configured to determine whether the violation is indicative of a computer security threat.

- 17
- 1 18. The method of claim 17, wherein the machine code representation comprises a
2 sequence of encoding fields, and wherein the part comprises a content of an
3 individual encoding field of the guest instruction.
4
- 1 19. The method of claim 17, wherein the part is indicative of an operator of the guest
2 instruction.
3
- 1 20. The method of claim 17, wherein the part is indicative of an operand of the guest
2 instruction.
3
- 1 21. The method of claim 17, wherein the predetermined location comprises a
2 predetermined register of the at least one hardware processor.
3
- 1 22. At least one hardware processor of a host system, the at least one hardware processor
2 configurable to:
3 in response to receiving a guest instruction for execution, determine whether executing
4 the guest instruction causes a violation of a memory access permission, wherein
5 the guest instruction executes within a guest virtual machine exposed by the host
6 system; and
7 in response to determining whether the guest instruction causes the violation, when
8 executing the guest instruction causes the violation:
9 write a part of a machine code representation of the guest instruction to a
10 predetermined location accessible to the computer security program;
11 suspend the execution of the guest instruction; and
12 in response to suspending the execution of the guest instruction, switch to
13 executing a computer security program, wherein the computer security
14 program is configured to determine whether the violation is indicative of a
15 computer security threat.
16

1 23. A non-transitory computer-readable medium storing instructions which, when executed
2 by at least one hardware processor of a host system, cause the host system to form a
3 computer security program configured to determine whether a violation of a memory
4 access permission is indicative of a computer security threat, and wherein the at least one
5 hardware processor is configurable to:
6 in response to receiving a guest instruction for execution, determine whether executing
7 the guest instruction causes the violation, wherein the guest instruction executes
8 within a virtual machine exposed by the host system; and
9 in response to determining whether the guest instruction causes the violation, when
10 executing the guest instruction causes the violation:
11 write a part of a machine code representation of the guest instruction to a
12 predetermined location accessible to the computer security program;
13 suspend the execution of the guest instruction; and
14 in response to suspending the execution of the guest instruction, switch to
15 executing the computer security program.
16