Nitro: Hardware-Based System Call Tracing for Virtual Machines

Jonas Pfoh, Christian Schneider, and Claudia Eckert

Technische Universität München, Munich, Germany
{pfoh,schneidc,eckertc}@in.tum.de

Abstract. Virtual machine introspection (VMI) describes the method of monitoring and analyzing the state of a virtual machine from the hypervisor level. This lends itself well to security applications, though the hardware virtualization support from Intel and AMD was not designed with VMI in mind. This results in many challenges for developers of hardware-supported VMI systems. This paper describes the design and implementation of our prototype framework, *Nitro*, for system call tracing and monitoring. Since Nitro is a purely VMI-based system, it remains isolated from attacks originating within the guest operating system and is not directly visible from within the guest. Nitro is extremely flexible as it supports all three system call mechanisms provided by the Intel x86 architecture and has been proven to work in Windows, Linux, 32-bit, and 64-bit environments. The high performance of our system allows for real-time capturing and dissemination of data without hindering usability. This is supported by extensive testing with various guest operating systems. In addition, Nitro is resistant to circumvention attempts due to a construction called hardware rooting. Finally, Nitro surpasses similar systems in both performance and functionality.

1 Introduction

Virtual machine introspection (VMI) lends itself very well to security applications [5]. This is, in part, due to the fact that security mechanisms running within the hypervisor are isolated from attacks that occur within a virtual machine (VM) and that the hypervisor maintains a complete and untainted view of a VM's system state.

In order to leverage the full potential that VMI provides, identifying and isolating the relevant guest operating system (OS) state information becomes crucial. This process requires some semantic knowledge about the guest and is referred to as the *semantic gap* issue [3]. Bridging this semantic gap has been classified into three fundamental *view generation patterns* [12]. One of these patterns relies on knowledge of the hardware architecture to derive semantic information about the guest OS. Making use of the hardware architecture allows one to construct mechanisms that are resistant to evasion attempts through a method called hardware rooting [13]. This makes hardware-based information extraction particularly interesting for security approaches that are intended to detect malicious activity within a monitored VM.

T. Iwata and M. Nishigaki (Eds.): IWSEC 2011, LNCS 7038, pp. 96–112, 2011.

One promising way of detecting such malicious activity is to monitor system calls. System calls facilitate communication between the kernel and user space within an OS and are interesting from a security perspective. System call traces may be used to classify the actions of a process as benign or malicious with machine learning approaches [9,6,14]. In addition, prevalent sandboxing environments, such as CWSandbox [7], Anubis [2], or the Norman Sandbox product line, incorporate system call or API monitoring to create their reports. Finally, particular system calls of interest are monitored in live forensic applications, such as those found in honeypot environments.

In this paper, we describe the implementation of our prototype VMI framework, *Nitro*, for hardware-based system call tracing and monitoring. Due to the properties of VMI, it is isolated from malicious activities within the VM and remains hidden from the guest OS. To our knowledge, Nitro is the first VMI-based system that supports all three system call mechanisms provided by the Intel x86 architecture and has been proven to work for Windows, Linux, 32-bit, and 64-bit guests. Moreover, this framework is flexible enough to feasibly support almost any OS built upon the x86 architecture. Capturing and disseminating data is done in real-time without hindering usability of the guest, as our performance tests show. Finally, Nitro is resistant to attempts at evasion due to hardware anchors. We will discuss the foundations of this approach, its implementation, and its properties throughout this work.

The remainder of this paper is organized as follows: We start by presenting some related work in Section 2. We go on to introduce the requirements for desirable properties of a VMI system in Section 3. The implementation of Nitro is detailed in Section 4, followed by a discussion of how it meets the aforementioned requirements in Section 5. In Section 6, we explain our performance evaluation and present the results, which we compare to a similar system called Ether [4] in Section 7. Finally, we draw our conclusions in Section 8.

2 Related Work

At the time of this writing, to our knowledge, there are three other systems that make use of virtualization extensions to implement systems that are capable of producing system call traces for security applications. The first system, Lares [11], was the pioneer in this area introducing a mechanism for creating arbitrary hooks within a Windows guest OS. Lares was developed on the Xen hypervisor and required drivers to be installed within the Windows guest to facilitate hooking. Since our system does not require any guest OS support at all, it achieves a much higher level of portability and robustness, as we will show. Thus, a detailed comparison between Lares and Nitro would not be very meaningful and is not provided in this paper.

In addition to Lares, the Ether system [4] provides the capability to produce system call traces. This system is also built upon the Xen hypervisor and takes a similar approach to that of Nitro. For a detailed comparison, please refer to Section 7.

Finally, HyperSleuth [10] also provides the ability to trace system calls, though the authors indicate that the "approach we use to trace system calls is thus inspired by Ether". For this reason we forgo a detailed comparison and direct the readers attention to our comparison between Nitro and Ether.

3 Properties

In Section 1 we outline the key properties of Nitro. These properties, a description, and the requirements to achieve each property are discussed within this section in a general manner. How exactly Nitro meets these requirements follows later in Section 5.

Guest OS Portability. Guest OS portability refers to a property that allows the same VMI mechanism to work for various guest OSs without major changes. Ideally, a guest OS portable VMI mechanism would work on any guest OS with no change, however we tolerate some minor configuration changes to the VMI mechanism, as long as the basic mechanics of that approach is shared among all guests.

In order to achieve guest OS portability, the underlying VMI mechanism may not rely on knowledge of the guest OS itself, but rather on knowledge of the virtual hardware specifications. For example, Jones et al. make use of the CR3 register in order to track processes [8]. How this register is to be used within the memory management unit (MMU) is specified by the x86 architecture, and all OSs running on this hardware and using the MMU must conform to these specifications. Thus, this basic method can be used to track processes in various guest OSs without change as long as the OSs support virtual memory.

Evasion-Resistance. An evasion-resistant mechanism is a mechanism which is impossible for an attacker to circumvent when *correctly implemented* and deployed in an *ideal system*. We define a correctly implemented mechanism as a mechanism that perfectly enforces the policy that it was designed to enforce with no flaws or errors. In the same manner, we define an ideal system as a system that perfectly implements its design and contains no flaws or errors. Since we know that these ideal properties are impractical, it may be possible to circumvent the mechanism if and only if such a flaw is found and exploited by a malicious entity. This is why we refer to this property as "evasion-resistant" rather than "evasion-proof". We begin this discussion by describing how a mechanism may be rooted in hardware.

In order to interpret the low-level binary state information of a virtual machine, the hypervisor must incorporate knowledge of the hardware architecture or the guest OS to bridge the semantic gap. As Pfoh et al. argue [12], an approach that relies on guest OS knowledge alone might be circumvented by attacks that change the guest OS architecture itself. For example, the manner in which the guest OS uses a particular data-structure may be manipulated by a malicious entity. This stems from the fact that this knowledge of the guest OS is in no way

bound to the running OS kernel. The fact that such attacks against VMI mechanisms have been successfully implemented recently [1] shows, that this threat is not of pure theoretical nature.

If, in contrast, the VMI mechanism bases its knowledge on information about the virtual hardware architecture, these attacks cannot be applied. The guest OS and all software running on it, including any malware, must play by the rules of the virtual hardware. An attacker has no means of changing these rules. Thus, this knowledge of the hardware specifications is bound to the hardware architecture. For example, if the hardware architecture specifies that a control register holds the address of a data-structure, there is nothing a malicious entity can do to circumvent this as the hardware will expect this to be the case in order to run correctly.

This argument can be expanded further to also cover other parts of state information as follows: If we can start at a feature of the virtual hardware specification (e. g., a register) and, from there, follow references in memory, thus building a chain to a critical data-structure, a malicious entity cannot modify that data-structure unnoticed. Figure 1 depicts such a chain. We will therefore refer to a portion of state information as being *rooted in hardware* if such a chain can be built [13].

Having introduced hardware rooting, we now describe the two requirements for an evasion-resistant VMI mechanism. First, the monitored or protected portions of the VM's state must be rooted in the virtual hardware, as described above. Second, each involved piece of VM state along the described reference chain must be protected such that it cannot be manipulated in violation of policy or that any change to such a piece of VM state is ignored by the guest OS. If both of these requirements are met the mechanism is evasion-resistant.

4 Implementation

This section describes the steps we took in implementing our prototype. Nitro is based upon the Linux Kernel Virtual Machine (KVM). KVM is split into two portions, namely a user application that is built upon QEMU and a set of Linux kernel modules.

The user application portion of KVM provides the *QEMU monitor* which is a shell-like interface to the hypervisor. It provides general control over the VM. For example, it is possible to pause and resume the VM as well as to read out CPU registers using the monitor. We modified KVM by adding new commands to the monitor to control Nitro's features. That is, all Nitro commands are input via this monitor.

These commands are then sent to the kernel module portion of KVM through an I/O control interface. The majority of Nitro is implemented in these kernel modules. Finally, the output is realized by making use of the *proc* filesystem. That is, Nitro creates a node in the *proc* filesystem and obtaining its output is as simple as reading from a file.

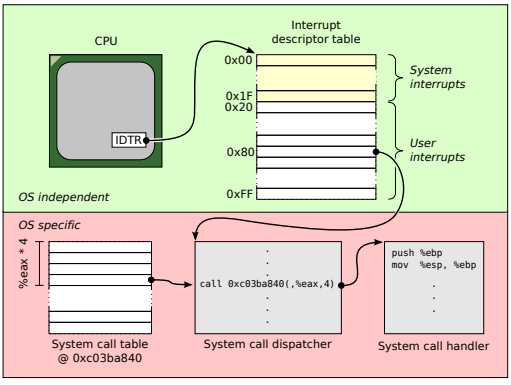

Fig. 1. The relationship between the Interrupt Descriptor Table Register (IDTR), the Interrupt Descriptor Table (IDT), and the system call dispatcher shows that the system call dispatcher is rooted in the IDTR through a chain that includes the IDT

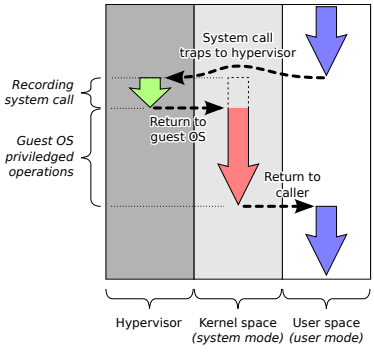

Fig. 2. Control flow of a system call that traps to the hypervisor

4.1 VMI Mechanisms for Trapping System Calls

In some cases the virtualization extensions provided by hardware manufacturers support trapping the specific event one is interested in, which makes the effort straightforward. However, it is often the case, especially for security mechanisms, that the hardware extensions do not support trapping the desired event. In these instances, we must indirectly induce a trap to the hypervisor. Finding these indirect methods for trapping desired events is often a challenge.

As it turns out, trapping to the hypervisor on the event of a system call is not supported on the popular Intel IA-32 (i. e., x86) and Intel 64 (formerly EM64T) architectures. In this case, we must find a way to indirectly cause the trap as discussed above. We do this by forcing system interrupts (e. g., page faults, general protection faults, etc) for which trapping is supported by the Intel Virtualization Extensions (VT-x). Hence, we have effectively created a mechanism for trapping system calls even though the hardware extensions do not natively support this. The resulting control flow is depicted in Figure 2. Since the three system call mechanisms are quite different in their nature, a unique trapping mechanism must be designed for each. These trapping mechanisms and their implementations are described below.

Interrupt-Based System Calls. System calls may be implemented as a user-defined interrupt. The x86 architecture handles interrupts through an Interrupt Descriptor Table (IDT). This IDT may have as many as 256 entries, each of which is 8 bytes long. The exact size of the IDT is stored in the IDTR along with the address at which the IDT resides in system memory. When an interrupt occurs, the hardware consults the IDT via the IDTR to determine the location for the appropriate handler and continues execution there as shown in Figure 1.

Intel's VT-x extensions allow one to trap system interrupts (interrupts 0 to 31) to the hypervisor, but they do not provide a mechanism for trapping user interrupts (interrupt 32 and above) which may be used for system calls.[1] This means that we must design a way to cause this user interrupt to generate a system interrupt.

We can achieve this by virtualizing the IDT, that is, we copy out the guest's IDT into the hypervisor. We must then manipulate the IDTR and trap all write accesses to it, thus disallowing any further manipulation. As the IDT size value stored in the IDTR is added to the base address to get the offset of the last valid byte of the IDT, we can set this size to $32 \cdot 8 - 1 = 255$. This leaves all system interrupts unaffected, however all attempts at invoking a user interrupt (i. e., interrupts greater than 31) will result in a general protection fault as the bounds of the IDT will have been exceeded. The advantage of this approach is that the IDT remains unaffected in memory, but is effectively ignored for user interrupts.

The next step is to trap all general protection faults to the hypervisor, which the virtualization extensions support natively. However, we must still determine the difference between general protection faults that we generated and those that occur naturally[2]. This can be done by inspecting the current instruction and determining whether or not it is the `int` instruction and whether the interrupt number is greater than 31.

If we identify the exception as being natural, we inject this exception into the guest and allow it to continue. However, if we recognize the exception to be caused by a user interrupt, we look at the interrupt number to determine whether we have trapped a system call. If this is the case, we collect data according to the rules specified for Nitro's data collection engine (see Section 4.3). In either case, the `int` instruction must be emulated using the IDT that we copied out of the guest and hand control back to the guest OS.

SYSCALL-Based System Calls. System calls may also be implemented using the `SYSCALL` instruction and its analogue counterpart `SYSRET`. Both of these rely on a set of MSRs, namely STAR_MSR, CSTAR_MSR, and LSTAR_MSR. Exactly which of these registers is used depends on whether the guest OS is running in legacy, long, or compatibility mode. Additionally, this mechanism can effectively be turned on and off by setting and unsetting the SCE flag in the Extended Feature Enable Register (EFER). Making use of either `SYSCALL` or `SYSRET` with the SCE flag not set results in an invalid opcode exception.

Forcing this mechanism to cause a system interrupt is then a matter of unsetting the SCE flag and setting the hypervisor to trap all invalid opcode exceptions, which is natively supported by the virtualization extensions. Once control has passed to the hypervisor, we must once again differentiate between natural exceptions and those caused by our introspection. This is achieved by looking at the violating instruction and if this instruction is *not* either `SYSCALL` or `SYSRET`,

[1] In contrast, AMD's SVM virtualization extensions do provide a mechanism for trapping user interrupts.

[2] We refer to exceptions that are not caused by our changes as *natural* exceptions.

we inject an invalid opcode exception into the guest OS and return control to it. However, if the violating instruction is, in fact, `SYSCALL`, Nitro collects the desired information, emulates this instruction, and returns control back to the guest OS.

In addition to emulating the `SYSCALL` instruction, Nitro must be capable of handling exceptions caused by the `SYSRET` instruction and emulating this instruction as well. This is due to the fact that the changes made to the EFER affect the `SYSRET` instruction in the same manner that they affect the `SYSCALL` instruction. Thus, use of the `SYSRET` instruction will also cause an invalid opcode exception and must be handled accordingly. In doing so, Nitro is also able to collect the return value of the invoked system call if the application requires this information.

SYSENTER-Based System Calls. Similar to `SYSCALL` and `SYSRET`, the `SYSENTER` and `SYSEXIT` pair of instructions also rely on a set of MSRs, namely SYSENTER_CS_MSR, SYSENTER_ESP_MSR, and SYSENTER_EIP_MSR. The values in each of these MSRs are copied into specific system registers upon a call to `SYSENTER`. Specifically and most interesting for the development of Nitro, the value of the SYSENTER_CS_MSR is copied into the CS register when `SYSENTER` is executed and an attempt to load the CS register with a null value results in a general protection exception. Hence, causing a system interrupt is a matter of saving the current value of the SYSENTER_CS_MSR register in the hypervisor and loading it with a null value. This will cause each `SYSENTER` operation to attempt to load a null value into the CS register, thus causing a system interrupt that the hypervisor can trap.

Once the hypervisor has trapped a general protection exception, differentiating between natural and forced exceptions is once more a matter of checking the current instruction at the time of the exception. If we come across a natural exception, as with the previous system call mechanisms, we inject the exception into the guest OS and allow it to continue. In the case that we come across general protection exception and the current instruction is `SYSENTER`, we collect the relevant data, emulate the instruction using the saved value of the SYSENTER_CS_MSR, and return control to the guest OS.

As with the `SYSCALL`/`SYSRET`-based system call mechanism, the change that we make to the guest in order to induce a system interrupt also affects the `SYSEXIT` instruction. Consequently, we must also emulate this instruction and with that, get a chance to easily extract the return value of the system call invoked.

4.2 Process Identification

It is always important to be able to determine which process produced a system call. This requires that we collect information which is unique to a process each time a system call is interrupted. Nitro collects the value of the CR3 register along with the value of the first valid entry in the corresponding top-level page directory. This allows us to identify a process due to the fact that the value in

the CR3 register (i. e., the address of the top-level page directory) is unique for a single process. In order to handle the case in which a newly created process receives a top-level page directory which is located at the same location of a previously destroyed process's top-level page directory, we also consider the first valid entry in the corresponding top-level page directory in order to create a truly unique identifier.

4.3 Collection of System Call Data

In our experience, different applications for system call traces depend on varying amounts of information. In some cases a simple sequence of system call numbers without arguments may suffice, while other scenarios may require detailed information including register values, stack-based arguments, and return values from a small subset of system calls. As we cannot foresee every guest OS type and possible application of system call tracing, Nitro does not deliver a fixed set of data per system call. Instead, it allows the user to define flexible rules to control the data collection during system call tracing in a fine-grained manner.

For example, the user can specify where exactly the guest OS stores the system call number (generally in the EAX register). Nitro can then extract this system call number along with a process identifier as described in Section 4.2. This information is often enough for certain machine learning techniques used for detection of malware or malicious behavior in processes [9,6].

In other instances, system call arguments or even dereferenced memory variables pointed to by arguments are crucial. To meet these requirements, Nitro's rules are expressive enough to account for both stack-based as well as register-based argument passing in addition to printing register values directly or dereferencing them. The syntax of such a rule takes the following format:

```
add_scmon_rule CONDITION_REG CONDITION_VAL ACTION_REG OFFSET ACTION,
```

where `CONDITION_REG` contains the name of the register that should be tested to determine whether information should be collected, `CONDITION_VAL` contains the value the `CONDITION_REG` should contain in order for further information to be collected, `ACTION_REG` contains the name of the register that contains the base value we are interested in, `OFFSET` contains the offset (positive or negative) from the `ACTION_REG` for the data we are interested in when collecting dereferenced values, and `ACTION` defines whether the `ACTION_REG` should be dereferenced as well as the format the output should take. This may result in printing or dereferencing the data as hexadecimal, integer, unsigned integer, or string. We provide a description of the rules in Backus-Naur Form in Appendix A. As an example, it is easy to specify a rule that dereferences and outputs the string being written every time a user process makes use of the write system call within a Linux guest. This rule would look as follows:

```
add_scmon_rule rax 4 rcx 0 derefstr
```

This rules-based method of requesting information makes Nitro very flexible and contributes to its OS agnostic nature.

Keeping the design goals for Nitro in mind, we collect only information whose location and format is defined by the hardware specifications or for which a rule is specified. However, the flexible design of Nitro allows an easy incorporation of guest OS specific knowledge in order to collect additional information about the calling process. We have successfully combined Nitro with other projects within our research group to include information such as process and user IDs into the output. However, we keep these projects separate in order to keep Nitro as simple and flexible as possible. This allows Nitro to be applicable in a greater range of applications. When combined with a memory analysis tool we call InSight, we are able to produce output as shown in Figure 3. This provides additional guest OS specific information, such as the type of descriptor being written to, while allowing Nitro to remain applicable of a wide range of guest OSs.

```
Jun 20 17:58:20: sys_write: unsigned int fd, const char
      __user *buf, size_t count
  fd: unsigned int: 0x3 → (socket) → [...]: (SOCK_STREAM)
      flags: ()
  buf: const char __user*: 0x7FFF702FF320 →
    buffer content hex (of size 107):
      47 45 54 20 2f 20 48 54 54 50 2f 31 2e 30 da 55 73 65
      72 2d 41 67 65 6e 74 3a 20 57 67 65 74 2f 31 2e 31 32
      20 28 6c 69 6e 75 78 2d 67 6e 75 29 da 41 63 63 65 70
      74 3a 20 2a 2f 2a da 48 6f 73 74 3a 20 67 6f 6f 67 6c
      65 2e 64 65 da 43 6f 6e 6e 65 63 74 69 6f 6e 3a 20 4b
      65 65 70 2d 41 6c 69 76 65 da da
    buffer content string:
      GET / HTTP/1.0
      User-Agent: Wget/1.12 (linux-gnu)
      Accept: */*
      Host: google.de
      Connection: Keep-Alive
  count: size_t: 0x6B
```

Fig. 3. Output of Nitro when combined with a memory analysis tool

5 Discussion and Evaluation

In most VMI-based mechanisms, performance overhead becomes a concern. For this reason, it is important to keep unnecessary traps to the hypervisor at an absolute minimum. In all the mechanisms described in Section 4, we make use of a system interrupt to facilitate the trap to the hypervisor due to the fact that system calls are not natively trappable.

For our implementation, we looked at the individual system call mechanisms and determined all system interrupts that each system call mechanism could

be made to produce and how to induce them. We then inspected all feasible solutions and considered them in terms of their impact on performance. For example, all three system call mechanisms can be made to produce a page-fault, however we passed on this for two reasons. First, page faults occur often (relative to other system interrupts) in regular system activity. This means that each page fault would result in a costly trap to the hypervisor to distinguish between forced and "natural" page faults, most of which would be natural. Second, this would essentially nullify any performance improvement that comes from using Extended Page Tables or Nested Page Tables.[3] In general, we strove for a system interrupt that occurs *infrequently* during normal operation and one whose use would not counteract performance enhancements in other parts of the system. This is how we came to the implementation and made our system viable for live collection.

Revisited: Guest OS Portability. Nitro is guest OS portable due to the fact that all three mechanisms described in Section 4 make sole use of hardware knowledge. This allows the mechanisms to work for any guest OS that is compatible with the x86 or the Intel 64 architecture. The IDTR and IDT as well as all the involved MSRs and their uses are specified by the hardware architecture and must be used in the way specified. That is, any guest OS must use these hardware mechanisms according to the specifications regardless of the guest OS.[4]

One potential hindrance for guest OS portability is the fact that how information is passed between kernel and user space is left to the OS designer. For example, some OSs are designed such that system call arguments are passed in registers, while others pass arguments on the stack. Nitro addresses this by providing the flexible set of rules described in Section 4.3. That is, the user can control which data is collected by specifying rules at run-time. This allows Nitro to be used across all guest OSs by simply changing the rule-set.

Revisited: Evasion-Resistance. We make the reasonable assumption that the hypervisor itself is secure. In addition, any components that reside within the hypervisor are safe from attacks originating from within a guest OS due to the hypervisor's isolation property.

While we have the aforementioned assumption with regard to the hypervisor itself, this alone is not enough. This is due to the fact that our VMI mechanisms make changes to the state of the guest VM. These state changes are clearly not protected by the isolation property as they take place within the guest OS itself. A malicious entity might simply revert the changes we made to the system state to circumvent our security mechanisms. For this reason we took special care to make sure that Nitro is evasion-resistant.

As stated in Section 3, evasion-resistance requires that the VMI mechanism is rooted in hardware and that each involved piece of VM state is protected

[3] These are hardware extensions implemented by Intel and AMD, respectively, to counter the performance degradation caused by using shadow page tables.

[4] Technically, an OS could also implement system calls entirely in software, but would then lack the privilege level feature of the CPU, leading to an insecure OS kernel.

against manipulation. Since the VMI mechanisms for the fast system call mechanisms and the interrupt-based mechanism differ slightly in this regard, they are discussed separately in the following.

In order to achieve an evasion-resistant VMI mechanism for fast system calls, it is rooted in either the SYSENTER_CS_MSR (`SYSENTER`-based) or the EFER (`SYSCALL`-based) as discussed in Section 3. In addition, the VMI mechanism may protect each of these registers from manipulation, which is directly supported by the virtualization extensions provided. This is enough to achieve evasion-resistance because Nitro's manipulation of the system call mechanisms are constrained to these registers. That is, due to the changes we made to the system, all fast system calls are trapped to the hypervisor and there is no way for a malicious entity to circumvent this without making changes to exactly those parts of the system that Nitro protects. Hence, this approach is both rooted in hardware and protects all involved pieces of VM state, resulting in an evasion-resistant mechanism.

Making the interrupt-based system call traps evasion-resistant is similar to the method described for fast system calls with one additional step. The mechanism is rooted in the IDTR and this register is protected against malicious manipulation by the hypervisor. In addition, a shadow copy of the original IDT is created within the hypervisor at boot time. This already is enough to achieve evasion-resistance as the changes to the guest OS are limited to this register. In addition, only the shadow IDT is referred to for each user interrupt. That is, any changes to the IDT within the guest OS do not affect the ability to trap user interrupts. In order to hinder this, a malicious entity would have to manipulate the IDTR directly or the shadow IDT within the hypervisor, both of which are protected with the help of the virtualization extensions.

6 Performance Testing

In this section, we present our general performance testing results for all guest OSs tested, which include: Windows XP SP2 (32-bit), Ubuntu Linux 9.04 Server (32-bit), and Ubuntu Linux 9.04 Server (64-bit). The tests were performed on an Intel Core 2 Duo processor at 2.4 GHz with 2 GB of RAM. We used a Debian Lenny (5.0.6 64-bit) host system for all tests. Finally, we used KVM 0.12.4 to act as the hypervisor.

These tests were performed by running benchmarks on the guest OSs once with Nitro disabled, then once with Nitro enabled and comparing these results. While the results themselves are of interest, we focus primarily on the amount of degradation observed because the degradation is a strong indicator for the overhead incurred by our system call tracing.

Throughout these tests, we made sure to test each mechanism. That is, we present tests that measure the performance of our mechanism for `SYSENTER`-based, `SYSCALL`-based, and interrupt-based system calls. It is important to note that the mechanism implemented at the time of this testing for the interrupt-based system calls function by redirecting the interrupt to a new gate descriptor

Table 1. Windows XP SP2 (32-bit) performance comparison between KVM/Nitro (`SYSENTER`-based) and Xen/Ether

Benchmark	*Xen/Ether*			*KVM/Nitro (`SYSENTER`)*		
	Tracing disabled	*Tracing enabled*	*Degradation*	*Tracing disabled*	*Tracing enabled*	*Degradation*
HTML Render [pg/s]	3.277	0.598	81.74%	2.826	**2.034**	**28.04%**
File Decryption [MB/s]	65.561	64.561	1.53%	64.697	**64.654**	**0.07%**
HDD [MB/s]	45.198	7.215	84.04%	46.545	**9.726**	**79.10%**
Text Edit [pg/s]	89.066	17.246	80.64%	84.743	**40.032**	**52.76%**
Image Decompression [MPix/s]	33.856	32.951	2.67%	33.364	**33.103**	**0.78%**
File Compression [MB/s]	2.737	2.677	2.19%	2.744	**2.741**	**0.10%**
File Encryption [MB/s]	15.821	15.515	1.94%	15.853	**15.826**	**0.17%**
Virus Scan [MB/s]	333.988	85.307	74.46%	314.118	**155.718**	**50.43%**
Mem. Latency [MemAcc/s]	6.735	3.580	46.84%	6.231	**3.782**	**39.30%**
PerformanceTest [score]	586.500	383.020	34.69%	628.700	**540.260**	**14.07%**

within the IDT, rather than emulating the `int` instruction. The following subsections present our results. All scores and times presented are a mean over three scores or runs.

Windows XP. In testing a Windows XP guest OS we made use of two commercial benchmarking products, namely PCMark05 from Futuremark and PerformanceTest from PassMark.[5] These tools perform various CPU, memory, disk drive, and graphics tests. Each make heavy use of system calls as is evidenced by the output of Nitro. PCMark05 returns a value for each performed test, while PassMark outputs a single combined score.

The standard deviation of all tests were negligible, except for the 'HDD' and 'Virus Scan' tests where we observed standard deviations of 6.3 and 61.0, respectively. We hypothesize that this is due to the fact that these are both disk I/O intensive tests. In any case, we present these results for the sake of completeness, however, due to their high deviation from the mean, we do not draw any conclusions from these values.

For these tests we were able to modify the virtual hardware such that the guest OS determined that the `SYSENTER` and `SYSEXIT` instructions were not available and thus resorted to the interrupt-based mechanism for system calls. This allowed us to test the performance of both `SYSENTER`-based (Table 1) and interrupt-based (Table 2) system call mechanisms.

It is interesting to note that across both sets of tests the degradation varies greatly from benchmark to benchmark, however the benchmarks with the lowest degradation ($< 10\%$) all perform some sort of compression, decompression, encryption, or decryption. Such functions are highly arithmetic and perform

[5] Available from `http://www.futuremark.com/products/pcmark05/` and `http://www.passmark.com/products/pt.htm`, respectively.

Table 2. Windows XP SP2 (32-bit) performance results with interrupt-based system calls on KVM/Nitro

	KVM/Nitro (interrupt)		
Benchmark	*Tracing disabled*	*Tracing enabled*	*Degradation*
HTML Render [pg/s]	3.05	2.28	25.12%
File Decryption [MB/s]	65.87	65.57	0.45%
HDD [MB/s]	46.06	9.13	80.17%
Text Edit [pg/s]	83.61	56.44	32.49%
Image Decomp. [MPix/s]	33.75	32.24	4.46%
File Compression [MB/s]	2.81	2.64	6.07%
File Encryption [MB/s]	16.10	15.23	5.42%
Virus Scan [MB/s]	325.34	102.63	68.45%
Mem. Latency [MemAcc/s]	6.23	4.67	25.00%
PerformanceTest [score]	623.16	557.66	10.51%

Table 3. Ubuntu Linux 9.04 Server performance results on KVM/Nitro

	Apache Compile Results		
Linux Guest OS	*Tracing disabled*	*Tracing enabled*	*Degradation*
32-bit Interrupt-based	168.989s	195.254s	15.54%
32-bit `SYSENTER`-based	167.916s	212.492s	26.55%
64-bit `SYSCALL`-based	179.166s	232.640s	29.85%

relatively few system calls since these arithmetic operations do not require OS support. We believe that this is the reason for the variation in degradation across the benchmarks. While the PCMark05 tests (the first nine benchmarks in Tables 1 and 2) are great for identifying to which degree an operation is affected by overhead in the system call mechanism, we feel that the results delivered by PerformanceTest give a better overall impression of the performance degradation in the guest OS as a whole.

Ubuntu Linux. For testing all Linux guest OSs we created a script that makes use of the 'time' command in Linux. Using this utility we measured the compile time of the Apache web server 2.2.16. The time utility makes use of the hardware clock and we verified beforehand that the hardware clock within the VM is consistent with the host system's hardware clock. We used this as a benchmark as it is resource intensive enough to show performance degradation and makes extensive use of system calls as is evidenced by the output of Nitro.

Presented in Table 3 are the test results when performed on a Ubuntu Linux 9.04 Server (32-bit) guest OS. As with our testing for the Windows XP SP2 guest, we manipulated the virtual hardware in order to be able to report results for interrupt-based and `SYSENTER`-based system call mechanisms. Considering these

results, we notice that the interrupt-based guest OS incurs less degradation than the `SYSENTER`-based guest. This is due to the fact that the mechanism in place for trapping the `SYSENTER` instruction requires emulating that instruction, while the mechanism in place for trapping the `int` instruction does not.

The testing process we used for a 64-bit Linux guest OS is identical to the processes we used for the 32-bit Linux guest with the obvious exception that we use Ubuntu Linux 9.04 Server (64-bit). One noteworthy difference between the 32-bit and 64-bit version of this operating system is that the 64-bit version makes use of the `SYSCALL`-based system call mechanism, making this the only test case that makes use of this instruction. These results are also presented in Table 3. Comparing the degradation of this guest to its 32-bit counterpart reveals that this OS incurred the most degradation among the Linux guests.

7 Comparison

We chose to compare our system to the Ether system [4] because Ether is the only other system to our knowledge (aside from HyperSleuth [10], which bases its system call tracing on Ether's approach) that supports some forms of system call tracing using VMI without having to install drivers or modules in the guest OS. Both in function and performance, Nitro surpasses Ether with regard to system call tracing and monitoring. In this section we discuss these differences in further detail.

Functional Differences. The largest functional difference between Ether and Nitro is the hypervisor that they are built upon. Ether builds upon the Xen hypervisor, while Nitro builds upon KVM. Nitro and Ether's system call tracing mechanism are similar in respect to the output they provide, though Ether's output is guest OS specific. That is, the output is Windows specific. Despite this, we tried Ether on further guest OSs in order to determine whether the underlying mechanism may be used for other guest OSs.

We tested both systems for functionality on Windows XP SP2 (32-bit), Ubuntu Linux 9.04 Server (32-bit), and Ubuntu Linux 9.04 Server (64-bit). Nitro proved functional on all tested platforms, while Ether proved 100% functional only on Windows XP SP2. While we expected Ether to be functional on Ubuntu Linux 9.04 Server (32-bit) because this OS uses the same system call mechanism as Windows XP SP2, the guest OS became very unstable and was not usable from a user's perspective when system call tracing was enabled. Finally, Ether was unable to provide any system call information for Ubuntu Linux 9.04 Server (64-bit) although the guest OS continued to run without issue. We believe that this is due to the fact that Ether fails to consider the `SYSCALL/SYSRET` mechanism for system calls completely. We see this as a major detractor as this limits the number of guest OSs for which system call tracing or monitoring will work.

Performance Comparison. We performed all presented tests on the same hardware and host OS as described in Section 6. In addition, we used the same Windows XP SP2 (32-bit) image for all tests to ensure the consistency of the

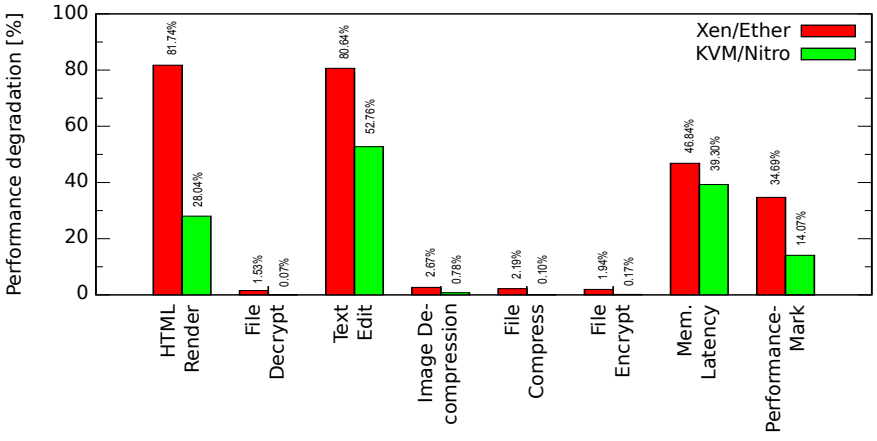

Fig. 4. Performance degradation of Xen/Ether and KVM/Nitro when tracing a Windows XP SP2 guest OS (ref. Table 1)

guest OS. Finally, for Ether we used the recommended Xen 3.1.0 as hypervisor. It is also important to note that we chose the specific benchmarks out of the PCMark05 suite due to the fact that these are the same benchmarks that Ether's authors used when testing their system originally. We felt it was important to include the same set of tests in the interest of equity.

Nitro and Ether are based on two different hypervisors, namely KVM and Xen. As our intentions were not to compare the performance between KVM and Xen but to compare the efficiency of the different implementations for system call tracing, we look at the relative performance degradation between the unmodified version of KVM and Nitro and compare this to the relative performance degradation between the unmodified version of Xen and Ether. This way we can measure the performance overhead incurred by each VMI implementation and do a fair comparison of Nitro and Ether.

As Ether only worked correctly for a Windows XP SP2 guest OS, we were only able to compare the performance of Ether and Nitro on this guest. These results are presented in Table 1 and Figure 4. Again, we do not draw any conclusions from the 'HDD' and 'Virus Scan' results due to their high observed standard deviation (this high standard deviation was observed for Xen/Ether as well as KVM/Nitro), however the results are present in Table 1 for the curious reader. Despite this, we see that Nitro outperforms Ether both on the absolute scores and the amount of degradation in all tests performed. In the benchmarks that focus on arithmetic operations (i. e., use relatively fewer system calls), for example compression and encryption tests, Nitro outperforms Ether only nominally. However in the case of HTML rendering, text editing, and memory latency, Ether's degradation is between 5 and 54 percentage points greater than that of Nitro. As mentioned in Section 6, while the PCMark05 benchmarks nicely reveal which specific operations incur the greatest degradation, the PerformanceTest benchmark is a better indicator of the overall degradation of the system. We see

that with this benchmark Ether's degradation is over 20 percentage points more than that of Nitro.

We hypothesize that Ether's greater degradation is primarily due to the fact that Ether forces a page fault interrupt to perform system call tracing. This adds additional overhead to a part of the hypervisor which is already responsible for incurring a large performance overhead. Additionally, this design decision effectively counteracts any benefits one might have from using Ether with a hypervisor which makes use of Extended Page Tables.

8 Conclusion

We have shown that Nitro is a powerful and flexible tool for system call tracing and monitoring. It supports all three system call mechanisms provided by Intel's x86 architecture for both 32-bit and 64-bit environments. In fact, we have successfully collected system call traces with Nitro for Windows, Linux, 32-bit, and 64-bit guests and we are confident that it will perform equally well for a variety of additional guest OSs. Further, the proven performance of our implementation allows the collection and dissemination of data in real-time. Finally, all of the VMI mechanisms presented have been shown to be evasion-resistant. That is, these mechanisms cannot be manipulated in a way which allows a malicious entity to circumvent system call tracing or monitoring. Its flexible and secure nature allows Nitro to be used effectively in a variety of applications, such as machine learning approaches to malware detection, honeypot monitoring, as well as sandboxing environments.

As Nitro builds upon KVM which is licensed under the GPLv2, we release the source code of our system under the same license. The source code is available at `http://code.google.com/p/nitro-kvm/`. We hope that this tool is useful for the community and will help to further security research.

Acknowledgements. The authors would like to thank Cornelius Diekmann for his contribution to this work by combining Nitro with other tools to help show its potential.

References

1. Bahram, S., Jiang, X., Wang, Z., Grace, M., Li, J., Xu, D.: DKSM: Subverting virtual machine introspection for fun and profit. In: Proc. of 29th IEEE Int. Symp. on Reliable Distributed Systems (SRDS 2010), New Delhi, India (October 2010)
2. Bayer, U., Kruegel, C., Kirda, E.: TTAnalyze: A tool for analyzing malware. In: 15th European Inst. for Computer Antivirus Research (EICAR 2006) Conf., Hamburg, Germany (April 2006)
3. Chen, P.M., Noble, B.D.: When virtual is better than real. In: Proc. of the 8th Workshop on Hot Topics in Op. Sys., p. 133. IEEE, Washington, DC, USA (2001)
4. Dinaburg, A., Royal, P., Sharif, M., Lee, W.: Ether: malware analysis via hardware virtualization extensions. In: Proc. of the 15th ACM Conf. on Computer and Communications Security, pp. 51–62. ACM, New York (2008)

5. Garfinkel, T., Rosenblum, M.: A virtual machine introspection based architecture for intrusion detection. In: Proc. of Network and Distributed Systems Security Symp., pp. 191–206 (2003)
6. Hofmeyr, S.A., Forrest, S., Somayaji, A.: Intrusion detection using sequences of system calls. Journal of Computer Security 6(3), 151–180 (1998)
7. Holz, T., Freiling, F., Willems, C.: Toward automated dynamic malware analysis using CWSandbox. IEEE Security & Privacy 5(2), 32–39 (2007)
8. Jones, S.T., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H.: VMM-based hidden process detection and identification using Lycosid. In: Proc. of the 4th Int. conf. on Virtual Execution Environments, pp. 91–100. ACM, New York (2008)
9. Kosoresow, A.P., Hofmeyr, S.A.: Intrusion detection via system call traces. IEEE Softw. 14(5), 35–42 (1997)
10. Martignoni, L., Fattori, A., Paleari, R., Cavallaro, L.: Live and trustworthy forensic analysis of commodity production systems. In: Jha, S., Sommer, R., Kreibich, C. (eds.) RAID 2010. LNCS, vol. 6307, pp. 297–316. Springer, Heidelberg (2010)
11. Payne, B.D., Carbone, M., Sharif, M., Lee, W.: Lares: An architecture for secure active monitoring using virtualization. In: Proc. of 2008 IEEE Symp. on Security and Privacy, pp. 233–247. IEEE, Washington, DC, USA (2008)
12. Pfoh, J., Schneider, C., Eckert, C.: A formal model for virtual machine introspection. In: Proc. of the 2nd ACM Workshop on Virtual Machine Security. ACM, New York (2009)
13. Pfoh, J., Schneider, C., Eckert, C.: Exploiting the x86 architecture to derive virtual machine state information. In: Proc. of the 4th Int. Conf. on Emerging Security Information, Systems and Technologies. IEEE, Venice (2010)
14. Rieck, K., Trinius, P., Willems, C., Holz, T.: Automatic analysis of malware behavior using machine learning. Tech. Rep. 18-2009, Berlin Inst. of Technology (2009)

A Nitro Output Rules Definition

We present the flexibility of our rules by expressing them in Backus-Naur Form.

rule	::=	**add_scmon_rule** <condition> <location> <action>
condition	::=	<register> <value>
location	::=	<register> <offset>
register	::=	rax | rbx | rcx | rdx | rsp | rbp | rsi | rdi
value	::=	[0,4294967295]
offset	::=	[-2147483648,2147483647]
action	::=	hex | int | uint | derefhex | derefint | derefuint | derefstr