ABSTRACT

This paper presents NumChecker, a new Virtual Machine Monitor (VMM) based framework to detect control-flow modifying kernel rootkits in a guest Virtual Machine (VM). NumChecker detects malicious modifications to a system call in the guest VM by checking the number of certain hardware events that occur during the system call's execution. To automatically count these events, NumChecker leverages the Hardware Performance Counters (HPCs), which exist in most modern processors. By using HPCs, the checking cost is significantly reduced and the tamper-resistance is enhanced. We implement a prototype of NumChecker on Linux with the Kernel-based Virtual Machine (KVM). Our evaluation demonstrates its practicality and effectiveness.

Categories and Subject Descriptors D.4.6 [Operating Systems]: Security and Protection-invasive software

General Terms Security

Keywords Kernel Rootkits, Virtualization, Hardware Performance Counters

1. INTRODUCTION

Kernel rootkits are formidable threats to computer systems. They are stealthy and can have unrestricted access to system resources. By subverting the operating system (OS) kernels directly, kernel rootkits are used by attackers to hide their presence, open backdoors, gain root privilege and disable defense mechanisms [8, 24].

Kernel rootkits perform their malicious activities in two ways: modifying the non-control data in the kernel data structures and hijacking the kernel control-flow to conceal resources from system monitoring utilities. In this work, we focus on the kernel rootkits that modify the kernel control-flow because the majority of kernel rootkits are of this type and they pose the most threat to system security. A recent analysis [16] indicates that more than 95% of Linux kernel rootkits persistently violate control-flow integrity. Control-flow modification makes the detection difficult because we do not know what the rootkits will modify. These rootkits may hijack the kernel static control transfers, such as changing the text of kernel functions or modifying the entries of the system call table. A representative example is SucKIT rootkit [22], which replaces the system call table with its own copy, and then uses its own system call table to redirect to the malicious system calls. The control-flow modifying rootkits may also hijack dynamic control transfers such as dynamic function pointers. Adore-ng [2] is a rootkit of this type. It manipulates the function pointers at the virtual file system layer to redirect the execution flow to malicious handler routines, which can hide information by filtering the data.

1.1 Detection Techniques and Limitations

There has been a long line of research on defending against control-flow modifying rootkits. Host-based rootkit detection techniques run inside the target they are protecting, and hence are called “in-the-box” techniques. For example, Rkhunter [6] and Kstat [5] detect the malicious kernel control-flow modifications by comparing the kernel text or its hash and the contents of critical jump tables to a previously observed clean state. The main problem with the “in-the-box” techniques is that the detection tools themselves might be tampered with by advanced kernel rootkits, which have high privilege and can access the kernel memory.

With the development of virtualization, the Virtual Machine Monitor (VMM) based “out-of-the-box” detection techniques have been widely studied. These techniques move the detection facilities out of the target Virtual Machine (VM) and deploy them in the VMM. The isolation provided by the virtualization environment significantly improves the tamper-resistance of the detection facilities because they are not accessible to rootkits inside the guest VMs. Most of the VMM-based rootkit detection techniques observe the static and dynamic kernel objects of a guest VM at the VMM level by directly acquiring the contents of the physical memory. However, there is a “semantic gap” between the external and internal observation. To extract meaningful information about the guest state from the low level view of the physical memory state, the detection tools require detailed knowledge of the guest OS implementation. For example, to retrieve the information of a guest VM’s process list, the detection tools need to know where this particular data structure is laid out in the guest kernel memory. The location may vary from one implementation to another. Acquiring this detailed knowledge can be a tough task especially when the kernel source code is not available.

In regards to security, because the knowledge of the guest OS that the detection tools rely upon is not bound to the observed memory state, these techniques are subject to advanced attacks that directly modify the layout of the guest kernel data structures [9].

1.2 Introducing NumChecker

To overcome the challenges that the current “out-of-the-box” detection techniques face, we propose an “execution-oriented” VMM-based kernel rootkit detection framework called NumChecker. Num-
Checker performs integrity checking at a higher level. It validates the whole execution of a guest kernel function without checking any individual object on the execution path. NumChecker models a kernel function with the number of certain hardware events that occur during the execution. Such hardware events include total instructions, branches, returns, floating point operations, etc. If the control-flow of a kernel function is maliciously modified, the number of these hardware events that occur during the execution will be different.

To count the guest hardware events from the host side, NumChecker utilizes the Hardware Performance Counters (HPCs), which exist in most modern processors as a part of the processor’s performance monitoring unit (PMU). The HPCs were originally used for performance tuning. NumChecker leverages them for the system security purpose. Because the events are automatically counted by the HPCs, the checking latency and the performance overhead are significantly reduced. Also, the security is enhanced because the HPCs count the events without a guest’s awareness, and they are inaccessible to a guest VM.

We implement a prototype of NumChecker on the Linux platform with the Kernel-based Virtual Machine (KVM) [4]. We evaluate NumChecker on a number of real-world kernel rootkits. The results demonstrate that NumChecker can efficiently detect all the kernel rootkits with very low cost.

The rest of this paper is organized as follows: Section 2 describes the background and related work. Section 3 gives the overview of our design. Section 4 presents the implementation details. The evaluation results are shown in Section 5. Section 6 is the conclusion. Additional implementation details are given in the appendix.

2. BACKGROUND AND RELATED WORK

2.1 Hardware Performance Counters

HPCs are a set of special-purpose registers built into modern microprocessors’ PMU to store the counts of hardware-related activities. HPCs were originally designed for performance debugging of complex software systems. They work along with event selectors which specify the certain hardware events, and the digital logic which increases a counter after a hardware event occurs. Relying on HPC-based profilers, the developers can easily understand the runtime behavior of a program and tune its performance. HPC-based profilers provide access to detailed performance information with much lower overhead than software profilers. Further, no source code modifications are needed.

HPC-based profilers are currently built into almost every popular operating system. Linux Perf [7] is a new implementation of performance counter support for Linux. It is based on the Linux kernel subsystem Perf_event, which has been built into 2.6+ systems. The user space Perf tool interacts with the kernel Perf_event by invoking a system call. It provides users a set of commands to analyze performance and trace data. When running in counting modes, Perf can collect specified hardware events on a per-process, per-CPU, and system-wide basis.

2.2 Related Work

Enhancing security with virtualization The use of virtualization technologies for enhancing system security has been studied for a long time. Garfinkel and Rosenblum [11] first introduced virtual machine introspection to detect intrusion. It leverages the virtual machine monitor to isolate the intrusion detection service from the monitored guest. XenAccess [17], VMMwatcher [14], and VMWall [23] are virtual machine introspection techniques using memory acquisition. These techniques obtain the guest states from the host side by accessing guest memory pages. As discussed in Section 1, to bridge the semantic gap, accurate kernel data structure layout or kernel symbols are required. Lares [18] monitors a guest VM by placing its hooking component inside the guest OS and protecting it from the VMM. These hooks would be triggered whenever certain monitored events were executed by the guest OS. This technique requires modification to the guest OSes, making it not applicable to close-source OSes like Windows.

Execution path analysis Patchfinder [21] is another closely related work that also uses execution path analysis for kernel rootkit detection. It counts the number of executed instructions by setting the processor to single step mode. In this mode, a debug exception (DB) will be generated by the processor after every execution of the instruction. The number counted during the execution of certain kernel functions will be analyzed to determine if the functions are maliciously modified. The vulnerability of this technique is that the counting and analysis facilities themselves might be manipulated by an advanced kernel rootkit which has the highest privilege and full access to kernel memory. From another perspective, running in the processor’s single step mode leads to very high performance overhead.

HPC-based integrity checking HPCs are originally designed for the purposes of performance debugging. Performing system security analysis is a new use of HPCs and has not been studied much. A scheme proposed in [15] uses HPCs for integrity checking of programs. It targets malicious modifications to the user space programs, and assumes the OS kernel is trusted. Also, it cannot be directly used for virtualization systems. Our design is to detect rootkits in a guest VM’s kernel space for virtualization systems.

3. NUMCHECKER OVERVIEW

3.1 Threat Model

We target a kernel rootkit which has the highest privilege inside the guest VM. The rootkit has full read and write access to guest VM’s memory space, so it can perform arbitrary malicious activities inside the guest VM’s kernel space. In order to hide its presence in the guest VM, the kernel rootkit modifies the kernel control-flow and executes its own malicious code. We assume that the VMM is trustworthy. And the rootkit cannot break out of the guest VM and compromise the underlying VMM [11, 14, 20].

3.2 System Call Analysis with HPCs

To detect control-flow modifying kernel rootkits, NumChecker focuses on validating the execution of system calls. System calls are the main interface that a user program uses to interact with the kernel. In order to achieve stealth, a common action that a kernel rootkit performs is to fool the user monitoring utilities (like ps, ls, netstat in Linux). These monitoring utilities retrieve the information about the system states by invoking some system calls. The
rootkits usually manipulate the normal execution of these system calls to prevent the monitoring tools from obtaining the correct information. For example, the Linux `ps` command will return the status of all the running processes. The system calls invoked by the `ps` command include `sys_open`, `sys_close`, `sys_read`, `sys_lseek`, `sys_stat64`, `sys_fstat`, `sys_getdents64`, `sys_old_mmap`, etc. To hide itself and other malicious processes, a rootkit modifies these system calls so that the information about the malicious processes will not appear in the list returned by `ps`. The modifications usually result in a different number of monitored hardware events from the uninfected execution. They are measured by NumChecker.

For a given system call, the number of hardware events that occur during the execution varies when different inputs are applied. To determine if an unusual number of events is caused by the malicious modification to the system call, the inputs to a monitored system call must be given. NumChecker invokes a monitored system call by executing a pre-generated test program in the guest VM. The counts of monitored events are then compared with those of the corresponding unmodified system call invoked by the same test program. By doing so, the noise from applying different inputs can be avoided.

Unlike the “in-the-box” techniques which perform the checking inside the monitored target (Figure 1(a)), or the “out-of-the-box” techniques which only depend on the observations from outside of the target (Figure 1(b)), NumChecker performs an “in-and-out-of-the-box” checking, shown in Figure 1(c): it first runs a test program inside the monitored guest VM. The test program will invoke monitored system calls (in-the-box). The host then accesses the HPCs to retrieve the guest state (counts of monitored events) from outside the guest VM (out-of-the-box). The combination takes advantage of both the meaningful information of the “in-the-box” checking and tamper-resistance of the “out-of-the-box” checking.

Because multiple programs are running concurrently in the guest VM, NumChecker has to identify the test programs. To overcome the semantic gap of the observation from the host side, NumChecker adds guest-transparent identifiers to the test programs to relate the guest VM’s state observed from the outside of the guest VM and the execution inside the guest VM. These identifiers are updated randomly and dynamically by the host. For more details, please see Appendix A.

4. NUMCHECKER IMPLEMENTATION

In NumChecker, we use KVM to build our virtualization environment. KVM is a full virtualization solution for Linux on hardware containing virtualization extensions that can run unmodified guest images. The processor with hardware virtualization extensions has two different modes: host mode and guest mode. Execution of virtualization-sensitive instructions in guest mode will trap to the host, which is called VM-exit. In this way, the host can manage the guests’ accesses to virtualized resources.

To profile the execution of system calls in a guest VM using HPCs, the profiler in the host should have the following capabilities: (1) it should be aware of the occurrence of system calls in a guest VM; (2) it should be able to trigger the HPCs. The existing HPC-based profiling tools cannot meet our design requirements because they are not able to capture the beginning and end of a system call in a guest VM. So the number of hardware events obtained by a profiling tool cannot be exactly pinned to the execution of a monitored system call.

To resolve this issue, NumChecker connects the profiling tool with the VMM, which is capable of intercepting system calls in the guest VM. NumChecker can be implemented with any HPC-based profiler. Our proof-of-concept design is based on the Linux `Perf`.

As shown in Figure 2, NumChecker has two main components: a lightweight module (A) in the host kernel between the KVM kernel module and the `Perf_event` kernel service, and a management program (B) running in the host user space. The kernel module of NumChecker performs two functions: first, it cooperates with the KVM to intercept monitored system calls in a guest VM (a). Second, it communicates with `Perf_event` kernel service to initialize, enable/disable, read, and close HPCs (b). The counted numbers are output to a log file (c). The management program is used to dynamically configure the module of NumChecker in the kernel by modifying the parameters through the `sysctl` system call (d).

A guest VM in KVM is seen as a single process from the host’s point of view. NumChecker calls the `Perf_event` kernel service to launch a per-process profiling task and enables the HPCs only when a monitored system call is run in the guest VM. By doing so, the counted events are exactly contributed by the execution of the monitored system call in the specific guest VM.

4.1 Two-phase Rootkit Detection

NumChecker rootkit detection has two phases, shown in Figure 3: in the offline profiling phase, system calls of the trusted guest OSes are measured; in the online runtime checking phase, system calls of a running monitored guest OS are measured and compared with that of the corresponding trusted OS.

4.1.1 Offline Profiling

In this phase, the test programs are executed in guest VMs with trusted OSes installed. The host logs in to the guest VM through the network between the host and the guest (for example, using SSH), loads executables of the test programs, and launches NumChecker. The configuration parameters specific to the monitored system call, such as the system call number and the type of hardware events measured, are passed to NumChecker. Then the test programs are executed in the guest VM. To improve the accuracy of the measurement, the execution of a test program is repeated several times. On the host side, the hardware events corresponding to the monitored system calls are counted. (More technical details about HPC-based measurement of a system call are presented in Appendix B.) When the measurement is complete, the system call interception is disabled. The results are stored as the “clean copy” to be used at runtime.

Since a system call may have various implementations in the OSes with different kernel versions, for each particular OS we need to create a separate “clean copy.” Generating a database containing clean copies of commonly used OSes is very fast because the number of kernel versions of commonly used OSes is limited. Unlike other comparison-based techniques, which take a long time to read large amounts of memory, for a given OS, NumChecker can profile one system call and create the “clean copy” in a few seconds.
Moreover, the offline profiling only needs to be performed once for a particular OS.

### 4.1.2 Online Checking

The execution path of a monitored system call is dynamically measured at runtime. The steps in this phase are similar to those in the offline profiling. The test programs are loaded when a guest VM is created. As mentioned above, to maintain consistency, the test programs used in the runtime checking are identical to the ones used to generate the “clean copy.” The system call profiling can then be dynamically invoked in either host-initial mode or guest-initial mode. For the host-initial mode, whenever the host administrator wants to launch a check, the host configures and enables NumChecker. Then the host logs in to the guest VM and executes the test programs in the guest VM. When the execution is done, the host turns off the system call interception.

Guest-initial mode is used for a guest user who wants to check if the OS is maliciously modified. In this mode, the guest user first sends a request to the host through the network. The management program in the host then checks the availability of HPCs and allocates unused ones to the guest. After the counters are allocated and ready to use, the host sends the acknowledgement to the guest. When the acknowledgement is received, the guest then runs the test programs to invoke monitored system calls.

### 4.2 Security Analysis

With the isolation provided by virtualization and the benefits of using HPCs, the “in-and-out-of-the-box” execution path analysis is very secure and tamper-resistant. Here, we discuss some possible attacks and show how they can be defended by our technique.

**Scenario #1:** The rootkit may try to tamper with the counting process. If the event counting is inside the guest VM, the kernel rootkit may disable the counters when its own code is executed and resume the counting when the control-flow returns to the normal execution. In this case, the malicious actions will not be detected since the counts remain the same as the unmodified execution. In our design, the hardware events are counted by the host. The HPCs are out of reach to the rootkits. Another way a rootkit could tamper with the counting process is that it may suspend the thread that runs the monitored system call and pass a pointer to another thread. The malicious code is then executed in the unmonitored thread where the events are not counted by NumChecker. However, suspending a monitored thread will cause a VM-exit that can be captured by the host. Malicious activities are suggested when this type of VM-exit is repeatedly observed during a check.

**Scenario #2:** The rootkit may tamper with the analysis process. Even though the counters are working properly and count all the true numbers, a rootkit may directly manipulate the analysis. Consider Patchfinder, the “in-the-box” execution path analysis technique, as an example. Since the counts are stored in the memory, the kernel rootkits who have full access to the memory can simply modify the actual counted number. For our VMM-based design, the counted numbers are read from HPCs by the trusted host and all the analyses are performed by the host. The guest kernel rootkits cannot interfere with the analyses because they do not have access to the host memory.

**Scenario #3:** The rootkit may try to predict the “good” number. Specifically, if the rootkit can predict the exact number of hardware events that occur during the execution of a system call, it could carefully modify the system call to generate the same number as the original one. However, given a system call, the number of hardware events generated in the execution depends on the inputs to the system call. In our design, the inputs to the monitored system calls are applied in the pre-generated test programs. A test program is used as the “secret key” in a particular check and it is updated dynamically by the host. A rootkit is not able to predict and generate a valid number of a monitored system call in a particular check because the number varies when different test programs are applied.

**Scenario #4:** The rootkit may undo modifications. A rootkit is used by attackers to provide long-term stealth for malicious activities. If a clever rootkit is aware of the occurrence of a check, it can undo modifications when the check is performed and activate itself again when the check is over. In NumChecker, the detection processes are running in the host without a guest’s awareness. The only thing the guest can see is the execution of a test program. However, from the guest’s point of view, the execution of a test program is no different from the execution of other programs. So a guest is not able to know when it is being monitored. Additionally, we can randomize the intervals between checks to avoid attackers’ prediction of the checking period.

## 5. EVALUATION

### 5.1 Detection Capability

To evaluate the effectiveness of NumChecker, we test our technique with eight real-world kernel rootkits on two different guest OSes. The host runs Ubuntu 11.10 with Linux kernel 3.0.16. The two guest OSes are Redhat 7.3 with Linux kernel 2.4.18 and Fedora core 4 with Linux kernel 2.6.11. Table 1 shows our experimental results. For each rootkit, we check the modifications it performs to five system calls, `sys_open`, `sys_close`, `sys_read`, `sys_getdents64`, and `sys_stat64`, with the corresponding test programs. Three hardware events, retired instructions (INST), retired returns (RN), and retired branches (BR), are monitored simultaneously for the execution of each system call. The percentages present the deviations of counts from uninfected executions.

To determine whether a system call is maliciously modified, one situation we must consider is false positives. Because of the complexity of an OS kernel, the noise is unavoidable. Even though identical test programs are applied, it cannot be guaranteed that the number of events is exactly the same for every single run. This noise can be reduced by increasing the number of times each test

---

**Figure 3:** Offline profiling phase (left-hand side) and online checking phase (right-hand side) of NumChecker rootkit detection.
program is run. In our experiment, each test program is repeated 500 times. We observe that the noise is less than 5% for the execution of a normal system call, no matter if the system load is light or heavy. So a deviation of more than 5% suggests a malicious modification.

From Table 1, we can see that in order to introduce their own functionality, the rootkits usually significantly modify the original system calls. The difference in the number of events between normal and infected executions is very notable. The Superkit rootkit modifies the system calls very heavily. The largest deviation is from the number of branches of the sys_open function, which is 1399.5%. The test on Sebek 3.2 gives smaller deviations. The largest deviation from the test on Sebek 3.2 is 18.8%, which is still much larger than the noise threshold of 5%.

### 5.2 Checking Latency

We perform the experiments of the checking latency on a PowerSpec platform with a 2.3GHz AMD Quad-Core Opteron 1356 CPU, which has 4 HPCs on each core. The host is running 32-bit Ubuntu 11.10 (kernel version 3.0.16) with 8GB RAM and 4-core configuration. The guest VMs are running 32-bit Redhat 7.3 (kernel version 2.4.18) and Fedora Core 4 (kernel version 2.6.11), with 512MB RAM and 1-core configuration.

Since NumChecker uses an "in-and-out-of-the-box" technique, the detection procedure has two separate parts, in the guest and host respectively. The total checking latency is the sum of the time for executing the test programs in the guest VM and the time for analyzing the counted numbers in the host. The checking latency depends on the number of test programs to be executed, the number of hardware events to be observed, and the specific guest OS version.

For each monitored system call, the analysis in the host takes 4.86 ms when three hardware events are observed simultaneously. Table 2 shows the execution times of each test program in two different guest VMs. To reduce false positives, each test program contains 500 iterations to repeatedly invoke the corresponding system call. The time is calculated from the total execution, and the numbers are averaged over 20 runs. Because test programs are very simple, the execution time is short. The average execution time of a test program on Redhat 7.3 and Fedora Core 4 are 45.6 ms and 59.5 ms, respectively.

A typical test of NumChecker checks all of the five system calls mentioned above with four corresponding test programs (sys_open and sys_close are checked in one test program). Table 3 presents the checking time of NumChecker for the Fedora Core 4 guest, and the comparison with other techniques. Because some results are from others’ experiments on different platforms, for each implementation, we list the CPU frequency and guest memory size, which are related more to checking latency. The first three techniques, Rhunter 1.2.8, Chkrootkit 0.48 [3], and Patchfinder, are host-based techniques running inside the target. MAVMM [12], VMwatcher, XenAccess, and OSck [13] are VMM-based techniques. For MAVMM, VMwatcher, and XenAccess, the checking time depends on the size of the memory to be examined because they require memory dumping. For VMwatcher, the checking time also depends on whether the kernel symbols are available. Examining a 512M raw window memory image takes 32 seconds while for Linux, the analysis can be finished within 500 ms.

NumChecker takes 24.3 ms in the host (5 system calls are analyzed) and 238 ms in the guest (4 test programs are executed). The total time to finish a typical test is 262.3 ms, regardless of the memory size of the guest VM.
6. CONCLUSION

In this paper, we present NumChecker, a VMM-based framework to detect control-flow modifying kernel rootkits in guest VMs. NumChecker performs the checking by validating the execution of system calls in the guest VM. The validation is based on the number of specified hardware events that occur during the execution of a guest system call. These hardware events are automatically counted by HPCs. We implement a prototype of NumChecker on Linux with the KVM virtualization environment, and our evaluation demonstrates its practicality and effectiveness.

7. ACKNOWLEDGEMENT

This material is based upon work funded by AFRL under contract No. FA8750-10-2-0101. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of AFRL.

8. REFERENCES

APPENDIX

A. IDENTIFYING TEST PROGRAMS

To avoid the noise from the execution of other processes, NumChecker needs to distinguish the system calls invoked by the test programs from ones invoked by other programs. This is done in two steps: identifying which system calls belong to the same process and identifying which process belongs to the test program.

To identify which system calls are invoked in the same process, the host can check either the CR3 register or the task descriptor pointer. To determine which process belongs to a test program, NumChecker generates an identifier for a test program with a special sequence of system calls. Specifically, a sequence of system calls (for example, one `sys_getuid` followed by two `sys_stat64` then three `sys_getuid` followed by four `sys_stat64`) is attached at the beginning of the test program. The process of the test program can be identified when NumChecker intercepts such a sequence in the same process. By carefully composing the sequence with some infrequently used system calls, it is almost impossible that another process can generate the same sequence by chance. The special system call sequence is only known by the host as a secret identifier, so an attacker in the guest VM is not able to use this to identify the test programs.

B. MEASUREMENT OF A SYSTEM CALL

System calls are implemented in two ways: interrupt-based system calls and sysenter-based system calls. The interception of guest system calls in both types is supported by processors with virtualization extensions, directly or with the help of other techniques [10, 19].

Figure 5 illustrates the procedure of measuring an interrupt-based system call in the guest VM. The test program invokes a monitored system call by executing an INT 0x80 instruction. This execution causes a VM-exit (arrow a in Figure 5) if the INT instruction interception is enabled in the host. When the system control is transferred to the host, NumChecker is launched to first determine whether the current system call needs to be monitored. If so, NumChecker passes configuration parameters to the Perf_event to initialize HPCs. These parameters include the counting mode, the type of hardware events, etc. After the HPCs are set up and right before entering the guest VM (b), NumChecker sends a signal to Perf_event to turn on the HPCs (c). Then the HPCs count the specified events when the system call is running in the guest kernel until reaching the end of the execution, which is an IRET instruction. System control is transferred to the host again (d). NumChecker turns off the HPCs (e) immediately after the CPU switches to host mode. The HPCs stop counting when the host code is being executed. The counted numbers are kept in the HPCs, NumChecker then reads the numbers through Perf_event (f) and outputs them into a log file. After that, the control is transferred back to the guest (g), and the execution of the program is resumed.

B.1 Handling Other Interceptions

When a guest VM is in kernel mode executing a system call, other virtualization-sensitive activities (such as I/O operations and external interrupts) of the guest VM may also cause a VM-exit (see block A in Figure 5). These activities interrupt the guest VM’s executions and will be intercepted by the host. If the HPCs keep counting when the VM-exit is being handled in the host, the events generated by the execution of the host code will be included in the final counts. To remove this noise, every time a VM-exit occurs during the execution of a guest system call, NumChecker suspends the HPCs (arrow h in Figure 5) by sending a disabling signal to Perf_event before the VMM handles the VM-exit. The HPCs are resumed (i) when the handling is finished.

Note that when an external interrupt returns, it also executes an IRET instruction. So if there are external interrupts taking place during the execution of a system call, more than one IRET instruction could be intercepted. It is necessary to find out the IRET corresponding to the monitored system call. This can be determined by checking the value of the stack pointer after returning from the interrupt. An external interrupt will return to the guest kernel space while a system call will return to the guest user space.

B.2 Handling Kernel Preemption

Traditional Linux kernels are not preemptible. When a task is running in kernel mode, it cannot be switched out until its completion, even though a higher-priority task is ready. Note that although the running task can still be interrupted by an external interrupt, a task switch cannot take place. In Linux 2.6 and later, a preemptible kernel option has been provided. This allows a higher-priority task to interrupt a running lower-priority task in the kernel. If kernel preemption is enabled, the HPC-based measurement becomes more complicated, shown in Figure 6. Whenever a monitored system call is suspended and switched out of the processor, the HPCs need to be suspended as well. Fortunately, a guest task switch can be intercepted (arrow j in Figure 6) by the VMM. When a task switch traps to the VMM, NumChecker disables the HPCs (k). It resumes the HPCs when the monitored system call is scheduled again (l). In this case, the execution of a system call is split into several “pieces.” These “pieces” can be determined by checking the task descriptor pointer of the task currently running on the CPU from the value of the ESP register. The “pieces” with the same task descriptor pointer belong to the identical system call.