Using Hardware Performance Events for Instruction-Level Monitoring on the x86 Architecture

Sebastian Vogl Claudia Eckert

Department of Computer Science
Technische Universität München
Munich, Germany
{vogls,eckertc}@in.tum.de

ABSTRACT

Full virtualization has become one of the basic technologies for the development of security applications. This is due to the fact that full virtualization provides important properties such as isolation and transparency that are essential for the development of robust security mechanisms. However, a fact that is often overlooked is that full virtualization also enables developers to make full use of the existing hardware features. By using these features in novel ways, it is possible to create new robust hardware-based security mechanisms.

In this paper we make use of the Performance Monitoring Counters (PMCs), which are available on most mainstream processors, to provide PMC-based trapping, a general concept for trapping hardware performance events to the hypervisor. We make use of this concept by proposing a novel approach to monitoring applications running within a virtual machine on the instruction-level from the hypervisor. In contrast to existing approaches, this course of action allows us to not only monitor all instructions of a program, but also enables us to limit the monitoring to specific instruction types. To demonstrate the possibilities of such an approach we implemented a shadow stack that protects the return addresses of functions running within a virtual machine from the hypervisor by only trapping call and return instructions.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection

Keywords

Performance Monitoring Counters, Instruction-Level Monitoring, Virtual Machine Introspection

1. INTRODUCTION

In the last years virtualization became more and more important for the field of information security and can nowadays, where many mainstream processors have integrated hardware support for virtualization, be seen as one of the fundamental technologies for the creation of security applications. This is due to the fact that full virtualization enables security applications to gain important properties such as isolation [7] or transparency [3], which makes the technology predestined for the development of robust security mechanisms. However, as important as these properties may be, one must not make the mistake to reduce the possibilities of full virtualization to these properties alone. A fact that is often overlooked is that full virtualization also enables us to make full use of the existing hardware features. By using these features in novel ways, it is possible to create new hardware-based security mechanisms that are reliable and robust [12]. In spite of this fact many of the available hardware features remain unexplored or unused. One of these hardware features are the Performance Monitoring Units (PMUs) of recent processor architectures.

In this paper we present a new hardware-based security mechanism for the x86 architecture. This mechanism makes use of the Performance Monitoring Counters (PMCs), which have been introduced with the Intel Pentium processor [8] and are nowadays available on almost all modern mainstream processors [9], to provide PMC-based trapping. That is, the mechanism is capable of trapping specific hardware performance events to the hypervisor. We use this capability to monitor instructions that are executed by programs running within a virtual machine (VM) from the host system. In contrast to existing hardware-based instruction-level monitoring (ILM) mechanisms, this approach enables us to monitor not only all instructions that are executed by a program, but also allows us to limit the monitoring to specific instruction types such as branch instructions. Due to this flexibility, the mechanism is well-suited as a basis for other security applications that require specific instructions to function. We demonstrate the latter through the implementation of a shadow stack that only traps `call` and `return` instructions to protect applications running within a VM from exploitation attempts that try to modify the return address of a function to execute malicious code. Since we leverage the potential of full virtualization to implement our security mechanism, it is not only flexible, but also guest transparent and evasion-resistant [13].

2. BACKGROUND & RELATED WORK

While PMC-based trapping is a new concept, there exist a lot of approaches that are capable of monitoring programs at the instruction-level. Since covering all of these approaches

would be well beyond the scope of this paper, we will in this section only provide a general overview of software-based approaches to instruction-level monitoring (ILM) and will on the hardware side focus on related hardware-based mechanisms that can similar to our approach be used to implement ILM from the hypervisor on the x86 architecture. Finally, we will also briefly summarize other research related to PMCs.

Software-Based ILM Approaches.

On the software side, ILM approaches often rely on one or a combination of three methods: Instrumentation, emulation, and instruction overwriting. In case of instrumentation, the monitoring code is added to the monitored program, which can be done during compilation, after compilation, or during run-time [1]. Frameworks that utilize the technique are Valgrind and Nirvana [1], for instance. In contrast, emulation-based approaches such as TEMU [16] make use of an emulator, in this case QEMU, to monitor program execution. Finally, approaches that are based on instruction overwriting replace existing instructions with the code required for monitoring or software breakpoints. This technique is often used by debuggers such as GDB or OllyDbg.

Hardware-Based ILM Approaches.

To the best of our knowledge, there currently exist three general hardware-based approaches that could be used to implement ILM from the hypervisor on the x86 architecture: Page-Fault (PF) based ILM [12, 15], Debug Register (DR) based ILM [12], and Trap Flag (TF) based ILM [3, 10]. However, none of these techniques can provide the flexibility to monitor specific instruction types. To show this each technique will be briefly described below.

Page-Fault (PF) based ILM. By making use of the page access bits of the shadow page tables or the extended page tables, it is possible to trap instructions that are either contained within (execute-disable bit) or try to read from (present bit) or write to (read-only bit) certain memory pages. Therefore by setting the desired access bits on every page, it is possible to trap all instructions or all instructions that involve memory operations on a guest system.

The main problem of using this approach for ILM lies in the fact that the mechanism is not working on an instruction basis, but rather on a page basis. Thus the mechanism is by design only able to trap all instructions contained within a certain virtual memory page or all instructions that have a memory operand accessing a certain virtual memory page.

Debug Register (DR) based ILM. The Intel x86 architecture provides four breakpoint DRs that can be used to trap instructions that are fetched from a specific memory address. By programming the DRs to contain the virtual memory address of every instruction that we want to monitor, we can implement an ILM mechanism.

In contrast to PF based ILM, this approach works on an instruction basis. However, the DRs are actually intended for setting hardware breakpoints. Therefore the only feature that the hardware supports in this case, is to raise an exception based on the locations specified in the DRs. The identification of these locations and the programming of the DRs has to be done in software. From this it follows that the complete logic that is required to implement a DR-based ILM mechanism, must actually be implemented in software. The role of the hardware is limited to the trapping itself, which the hardware cannot fulfill in all situations, due to the fact that hardware breakpoints that are placed directly after a `POP SS` or `MOV SS` instruction may not be triggered [8]. Thus DR-based ILM is not evasion-resistant [13].

Trap Flag (TF) based ILM. Using the TF for ILM is a very common approach which is, for example, adopted by Ether [3] and MAVMM [10]. The TF is a system flag, which will - if set - lead to the generation of an exception after every instruction that is executed by the processor. Therefore this hardware mechanism is the only one of the here presented mechanisms, that is actually intended for ILM. Besides, single-stepping the TF can in theory also be used to implement branch monitoring by combining it with the single-step on branches flag (BTF). However, this functionality cannot be used in practice, since the TF is modified by some guest operating systems (OSs) and the processor [8]. Once cleared, the processor will no longer generate exceptions and hence control will be lost. This also complicates single-step monitoring with the TF, but when every instruction is trapped, it is at least possible to reset the TF in case it was cleared between two instructions.

The whole problem arises due to the fact that the TF cannot be protected by the hardware. This means that there is no hardware mechanism which we are aware of that could be used to raise a signal if the TF is modified. Thus a TF-based ILM mechanisms is not only unreliable, but also not evasion-resistant, since every process has access to its own TF and can therefore arbitrarily manipulate it.

Overview of related work involving PMCs.

ReVirt [5] and Aftersight [2] use the PMCs to be able to replay program executions within a VM. Du et al. [4] as well as Nikolaev and Back [11] provide solutions to PMCs-based profiling within VMs, while Malone et al. [9] propose to use PMCs for integrity checking of programs. Finally, there are many frameworks and libraries that provide a common interface to PMCs on different architectures. Examples include perf_events for Linux, the Performance Application Programming Interface (PAPI) for Linux, Solaris, FreeBSD and other OSs, and the Windows Management Instrumentation (WMI) for Microsoft Windows.

3. PMC-BASED ILM

In this section we will present our approach to ILM from the hypervisor on the Intel x86 architecture. In contrast to existing hardware-based approaches, it allows to select specific instruction types for monitoring. For this purpose, we will first introduce PMC-based trapping, which is a general concept that can be used to monitor hardware performance events from the hypervisor. We will then depict how this general technique can be applied to realize ILM. However, before we will go into details about PMC-based trapping, it is important to know that even though almost all mainstream processors have Performance Monitoring Units (PMUs) [9], their properties and possibilities may vary. Therefore some of the facts described within this section, may only be valid for the Intel x86 architecture.

3.1 The Intel x86 PMCs

Most mainstream processors provide a PMU that enables applications to measure the performance of other applications or their own performance by monitoring the occurrence of specific hardware events. To reduce the overhead of the performance measurement and the lines of code that have to be added to an application that uses the PMU [14], an

application usually does not monitor these events directly. Instead the PMU provides so-called Performance Monitoring Counters (PMCs) that count the occurrence of specific hardware events and can be accessed by the applications. To illustrate this mechanism, consider a PMC, for example, that counts the number of instructions that have been executed by the processor. By reading this counter multiple times and storing its value, an application can deduce how many instructions were executed in a certain period of time.

On the Intel x86 architecture, PMCs are model-specific registers (MSRs) that can in addition to the `RDMSR` and `WRMSR` instructions be accessed with the `RDPMC` instruction. Depending on the type of the PMC, the event that is counted by it is either fixed or programmable. While fixed PMCs count a specific hardware event that cannot be changed, programmable PMCs can be set to count one of the supported hardware events. For this purpose there exists another MSR, the performance event select (PES) MSR, that determines the event that is counted by its corresponding PMC. Besides the event itself, the PES MSR of a PMC also provides additional controls that can be used to influence the counter. For example, it is possible to specify if an event is only counted at a certain processor privilege level and if the PMC should generate an interrupt when it overflows.

The number of fixed and programmable PMCs as well as the type of events that can be counted by a programmable PMC depend on the specific processor microarchitecture. On Sandy Bridge-based processors, for example, there are three fixed and four programmable PMCs [8]. When it comes to the countable events, it is necessary to distinguish between architectural events and non-architectural events. While architectural events are available on all x86 processor architectures, the number and type of non-architectural events depend on the specific processor microarchitecture. If we once more consider the Sandy Bridge microarchitecture, we find that there exist more than 200 non-architectural events in addition to the seven architectural events that are available across all Intel x86 microarchitectures [8].

3.2 PMC-based Trapping

The idea behind PMC-based trapping is to trap the occurrence of hardware performance events that are counted with the help of programmable[1] PMCs to the hypervisor such that these events can be used to implement security mechanisms. As in the case of other security applications that try to monitor certain events within a VM from the vantage point of the hypervisor, the realization of such a mechanism requires two steps: First, it is necessary to force the hardware to generate a *signal* whenever the event occurs that we want to observe. Second, this signal must lead to a VM Exit that *transfers control* to the hypervisor.

In case of PMC-based trapping, the first step requires us to make sure that a PMC, which was set to count a specific hardware event, generates a signal when a certain number of events occurred. Since a PMC produces an interrupt on overflow, given that the corresponding flag was set in the PES MSR that controls the PMC, we can emit a signal if we force the PMC to overflow. This can be achieved by setting the initial value of the PMC to `MAX_PMC_VALUE` - X + 1, were X is the number of events that should occur before the overflow. For example, to cause an overflow after every counted event, we set the PMC to its maximum value. Thus the PMC will overflow when the next event is counted.

The interrupt signal that is emitted by a PMC on overflow depends on the setting of the local Advanced Programmable Interrupt Controller (APIC). Amongst other things, the local APIC allows us to force the delivery of a Non-Maskable Interrupt (NMI). Emitting a NMI has two advantages. First, the processor will handle the interrupt immediately, which will reduce the time that passes between the counter overflow and the moment the interrupt is handled by the processor. This is important due to the fact that it is possible that more than one event occurs during the time it takes to deliver the interrupt. For a more detailed discussion of the problem we refer the reader to Section 3.4.1.

Second, a NMI leads to a VM Exit if the appropriate flag is set within the pin-based VM-execution controls of the x86 architecture. Thus by using a NMI we can force a VM Exit and thus realize a control transfer to the hypervisor.

3.3 Transparency and Evasion-Resistance

While the steps described in the last section allow us to trap hardware performance events to the hypervisor, it is not yet clear if PMC-based trapping is evasion-resistant [13] and transparent [3]. But, before we can analyze these important security-relevant properties in the context of PMC-based trapping, it is necessary to clarify what we mean by a transparent mechanism. This is due to the fact that complete transparency within virtualized environments is difficult, if not impossible, to achieve as Garfinkel et al. [6] rightfully argue. Therefore we use the term *guest transparency* within this paper to refer to a hypervisor-based mechanism that can only be detected through timing attacks.

To achieve both evasion-resistance and guest transparency, it is necessary that none of the hardware registers that are used by a PMC-based trapping mechanism can be accessed from within the guest. Since the processor-based VM-execution controls on the x86 architecture provide the possibility to cause a VM Exit in case a MSR is read or written or a `RDPMC` instructions is executed, read or write accesses to the PMCs can be intercepted by the hypervisor. In addition, because all control structures that are related to PMCs are MSRs as well, attempts to access or modify the PMC control structures can also be intercepted. Thus PMC-based trapping is evasion-resistant and guest transparent provided that the hardware does not contain any flaws and implements the above mentioned mechanisms correctly [8, 13].

3.4 PMC-based Trapping for ILM

PMC-based trapping can be directly applied to ILM by making use of hardware performance events that are related to instruction execution. Since modern microarchitectures support events that allow to count specific instruction types, the resulting ILM approach is capable of selecting these instruction types for monitoring. To provide the reader with a better understanding of the type of events that are available, Table 1 shows a small subset of the non-architectural events related to taken branch instructions that are about to retire and are available on modern Sandy Bridge processors. In this context the term "retire" stands for instructions that have been executed by the CPU and whose changes will be committed to the architecture in the correct order in which they appear within the instruction sequence [8].

Architectural events that are important for ILM and are

[1] A similar approach could also be realized using fixed PMCs.

Event Type	Description
ALL_BRANCHES	All branch instructions
CONDITIONAL	All conditional branch instructions
NEAR_CALL	All near call branch instructions
NEAR_RETURN	All near return branch instructions
FAR_BRANCH	All far branches

Table 1: Non-architectural events related to retired branches available on recent processors.

available across all architectures are all retired instructions and all taken branches that are about to retire. Thus even if a microarchitecture would not support any other hardware event, it would in any case be possible to trap all instructions or limit the trapping to all branch instructions.

3.4.1 Counter Overflow Issues

There is the possibility that more than one event occurs before the PMC overflow interrupt is received. The reason for this phenomenon is latency within the microarchitecture in combination with the speed of modern processors. Due to the latter it is likely that multiple instructions retire in a very short period of time, which also means that multiple monitored events could occur at nearly the same time. The PMC that counts these events will therefore be increased and will on an overflow generate a signal. This signal will then be forwarded to the local APIC that in turn will raise the selected interrupt. Because of latency, it is hence possible that more than one event occurs during the period of time between the PMC overflow and the delivery of the interrupt.

How many events can occur before the interrupt is delivered depends on the event that is monitored. The closer events appear to each other within the instruction stream, the higher are the chances that they will occur together before the interrupt is received. If we implement a single-stepping mechanism with the help of a PMC by forcing an overflow after every instruction executed by a system, for example, it is very likely that more than one instruction will be executed before the interrupt is received. In fact, during our experiments with such a PMC-based single-stepping mechanism, we found that on the average about six instructions were executed before the interrupt was received.

Notice that similar issues could arise if PMC-based trapping is applied to other areas. How this problem can be solved must be considered on a case-by-case basis. In the following we will provide a solution for PMC-based ILM.

3.4.2 Instruction Reconstruction (IR)

Because of the speed of modern processors and the latency within the microarchitecture, it is possible that we miss instructions that we want to monitor. However, the fact that we missed an instruction will not go unnoticed. On the contrary, the PMC that is used to count the instructions will tell us exactly how many instructions were missed, since it continues counting even after an overflow occurred. Therefore we know how many instructions we missed between the occurrence of the last interrupt and the current interrupt. This effectively reduces the general problem of recovering all missed instructions to the smaller problem of recovering all missed instructions that lie on the execution path from the last to the current interrupt. A possible solution to this problem is to reconstruct this execution path, reanalyze all executed instructions, and filter the instructions that we missed. To achieve this we will save the instruction pointer (IP) of the VM every time a monitoring related interrupt is received. This allows us to use the IP of the last interrupt as a starting point for the IR. To reconstruct the instructions, we will then sequentially decode all instructions that follow this starting point until we reach the current IP. A remaining problem are control transfer instructions that we encounter along the way, since the target of the control transfer may depend on memory operands that may have been overwritten in the meantime. However, since the control transfer instructions that we encounter were already processed during normal execution, we can make use of another hardware feature to recover the target locations of these instructions: The Last Branch Record Stack (LBR).

As the name suggests, the LBR is a stack that contains the last instructions that caused a control transfer. More precisely, the LBR consists of MSR pairs, where one MSR, the FROM MSR, contains the virtual address of the instruction that caused the control transfer and the other MSR, the TO MSR, contains the virtual address of the target of the control transfer. Therefore the LBR can be used to decide, if a branch was taken and what the resulting IP value was.

The size of the LBR depends on the processor that is used. Recent processor families usually provide a LBR that consists of 16 MSR pairs. The top of the LBR is indicated by a special MSR, the so-called top-of-stack pointer (TOS) MSR. Once the TOS reaches its maximum value, it will wrap around and the oldest entry on the stack will be overwritten. Therefore the number of branches that can be reconstructed with the help of the LBR, depend on its size. However, we do not expect this to be an issue in practice, since the size of the LBR was always sufficient for IR during our experiments. But even if we had to record more branches than the LBR can hold, this is not a problem, because it is also possible to use the Branch Trace Store (BTS) instead of the LBR, which is basically a LBR in memory and can therefore hold as many branches as memory is reserved. Further in contrast to the LBR, the BTS can be programmed to generate a signal before it overflows, which means that branches can always be processed before they get overwritten [8].

By using the LBR it is finally possible to reconstruct the complete instruction stream by saving the TOS of the LBR in addition to the IP on every monitoring related interrupt. While moving through the execution path starting from the last IP we can then increase the last TOS whenever a control transfer instruction is encountered that is recorded on the LBR and continue sequential decoding from the destination address. This will eventually lead us to the current IP.

4. EXPERIMENTS & ANALYSIS

In this section we will describe the prototype that we implemented and the experiments that we conducted with it. In the process we will also provide more details about the shadow stack that we implemented to demonstrate the potential of PMC-based ILM. Finally, we will analyze the results that the experiments produced.

4.1 Implementation

To be able to conduct experiments and to show the practicability of our approach we implemented a prototype that is based on KVM. Similar to KVM this prototype is split into two components: A user and a kernel space component. The kernel space component is integrated within the

KVM kernel modules and is responsible for the actual monitoring. Currently this component is capable of trapping any PMC event that the microarchitectures supports. On top of that, we also implemented single-step monitoring based on the TF for the purpose of comparing our monitoring mechanism with an existing hardware-based monitoring mechanism. We chose to use the TF for a comparison for two reasons. First, it is the only hardware-based mechanism that is actually intended for ILM and second, it is probably the hardware-based ILM mechanism that is used most often.

The user space component of the prototype is built upon QEMU, which is the user application part of KVM. While the kernel space component is only responsible for trapping, the task of the user space component is to process the trapped instructions. Although this approach reduces the performance of the prototype, since data frequently needs to be transferred from kernel to user space, it results in a cleaner design where only tasks are performed within the kernel modules that cannot be done in user space.

Note that the current prototype works on a per process basis, which means that only user selected processes are monitored, instead of the complete system. To achieve this, the prototype hooks the system calls for process creation and termination using the DRs. Since the placement of these hooks depends on the guest OS, the current prototype only supports Linux guests. We are currently working on a Windows version that will be completed in the near future.

4.2 Experiments

Our current prototype is tailored towards simple experiments so that we can demonstrate the potential of PMC-based ILM mechanisms. To give an initial impression about the performance of a PMC-based approach, we compared PMC-based ILM with traditional TF-based ILM by monitoring the user space part of common Linux applications on an Ubuntu 11.04 Server 64-bit VM. In particular we monitored the execution of `ls`, `tar`, `cat`, and `gcc`. Thereby `ls` was set to display the contents of the `/usr/bin` directory, which contained 597 files, while the remaining three programs were used to list, compress, or respectively compile a simple 'Hello World' program that consisted of 10 lines of code. We decided to limit the monitoring within the VM to the user space part of the applications to avoid differences in execution time due to guest kernel code that may be executed in case of one application run, but not in case of other runs. While constraining the monitoring to the user space part of an application was straight forward in the case of PMC-based ILM, we were in case of TF-based ILM forced to detect switches from kernel to user space by monitoring for changes in the virtual address of the IP. This is due to the fact that there is no flag as in the case of the PES MSR that provides this functionality in hardware. But since the TF was disabled by the guest OS when a switch from kernel to user space occurred, we had to detect this behavior in any case to be able to reenable the TF.

All experiments were conducted on an Ubuntu 11.10 host system with 8 GB RAM and an Intel Core i7-2600 CPU that was running at 3.40 GHz. To measure the average slowdown factor of ILM on the applications, we executed the same program with the same parameters multiple times and recorded the total (wall clock) execution time for each program run. In the process we made use of the PMCs to limit the monitoring to specific instruction types and recorded how this limitation affected the slowdown factor. In particular we set the PMCs to monitor all instructions (with and without IR), all branch instructions, and all `call` and `return` instructions. The latter allowed us to implement a shadow stack that will be described in more detail in the next section. The results of the experiments are shown in Table 2.

Mode	ls	tar	cat	gcc
PMC ALL&IR	755 (18s)	1002 (3.0s)	334 (0.6s)	1263 (92s)
TF ALL	310 (7.0s)	415 (1.2s)	142 (0.3s)	545 (40s)
PMC ALL	273 (6.5s)	403 (1.2s)	126 (0.3s)	435 (32s)
PMC Branch	163 (4.0s)	259 (0.8s)	81 (0.2s)	281 (21s)
Shadow Stack	95 (2.0s)	196 (0.6s)	31 (0.1s)	212 (15s)

Table 2: The average slowdown factor of different monitoring modes on common Linux applications.

4.3 Shadow Stack

By using PMC-based trapping, we are able to monitor specific instructions types. To demonstrate this possibility we implemented a shadow stack on the hypervisor level that is capable of verifying if the return address of a function was modified during execution by trapping `call` and `return` instructions. Every time a `call` instruction is executed by the monitored process, the return address of the called function, which is the address of the instruction that immediately follows the `call` instruction, is pushed on the stack. When the process later on executes a `return` instruction, the address on top of the shadow stack is compared with the new value of the IP. Since a function should return to the location immediately after the `call` instruction that invoked it, both values should be equal, because the return address of the last `call` instruction is on top of the stack. Therefore, if the values do not match the return address was modified.

Since attackers often try to modify the return address of a function in order to divert the control flow of a program, the shadow stack is capable of detecting exploitation attempts on applications running within a VM from the hypervisor level. In addition, because our shadow stack implementation solely relies on PMC-based trapping to function, it is evasion-resistant and guest transparent.

The correct functioning of the shadow stack was verified with the help of simple buffer overflow exploits that tried to overwrite the return addresses of multiple vulnerable functions. All exploits were reliably detected. Further no false positives were observed during the execution of the performance experiments, whose results are shown in Table 2.

4.4 Analysis

Flexibility. As the first column of Table 2 shows, PMC-based ILM allows to select specific instructions for monitoring and thus provides more flexibility than existing hardware-based ILM mechanisms on the x86 architecture. In addition, monitoring can also be limited to kernel and/or user space by using the appropriate flags within the PES MSR that belongs to a PMC. Nevertheless, it is possible to monitor all executed instructions by reconstructing the complete instruction stream with the help of the LBR. Finally, the implementation of a shadow stack demonstrates that the events, which are supported by current processors, can be used effectively to create security modules on the hypervisor level, which are guest transparent and evasion-resistant.

Performance. The experiments that were conducted so

far can only give an indication of the performance of a PMC-based ILM approach. In spite of that, the current results are promising and indicate that such an approach can, when used without IR, be much faster than existing hardware-based ILM approaches, because a PMC-based approach is not forced to cause a VM Exit after every executed instruction. This performance gain is especially visible in case of applications that only require access to a certain subset of the executed instructions such as the implemented shadow stack, which performed in all conducted experiments at least twice as fast as the TF-based mechanism.

PMC-based single-stepping with IR on the other side was clearly slower than TF-based single-stepping. This has two main causes. First, as explained in Section 2, the TF is cleared by the processor in certain situations such as the execution of an interrupt handler, while a PMC-based approach monitors the complete execution. Thus PMC-based ILM with IR actually monitors more instructions than TF-based ILM. Second, since the current prototype uses the QEMU disassembler to disassemble instructions during IR, every instruction is individually processed and completely disassembled. However, disassembling every instruction completely is actually unnecessary, since for most instructions we only require their length and opcode to continue IR.

Even though PMC-based ILM can provide better performance than existing hardware-based ILM mechanisms, the current results show that such an approach still leads to a significant overhead. To reduce this overhead we suggest the following measures that we leave for future work.

First, the current prototype always sets the PMCs to their maximum value to force an overflow after every event. This guarantees that events are delivered with as little delay as possible. However, this may not be necessary for all applications. In case of the shadow stack, for instance, we do not need to force a VM Exit after every `call` or `return` instruction, since the LBR allows us to process a group of those instructions without having to use IR. By reducing the overflows, however, we will reduce the slowdown factor by almost the same amount. For example, when we set the PMC-based branch monitoring to only force a VM Exit after ten branches instead of after every branch, the average slowdown factor stated in Table 2 was reduced by 88%.

Second, by using Precise-Event-Based Sampling[2] it is without a VM Exit possible to store the values of all general purpose registers on a PMC overflow in memory. Thus this mechanism would allow us to reduce the number of VM Exits, while still being able to record events on a fine-grained basis in memory, which can then be inspected on the next VM exit. However, we did not implement this mechanism so far, since it requires the allocation of memory within the guest. While the allocation itself can be achieved by manipulating the page tables of the guest, protecting this memory area from the hypervisor is an interesting challenge, because the memory area must be writeable.

5. CONCLUSION

In this paper we presented the concept of PMC-based trapping. We applied this concept to implement a hardware-based ILM approach that is capable of monitoring all instructions that are executed by a process as well as only specific instruction types. This functionality allowed us to implement a shadow stack that shows the potential of the approach and demonstrates the performance gain that can be achieved in comparison to traditional hardware-based ILM mechanisms by using it. However, the experiments that were conducted so far can only give an indication of the possibilities of a PMC-based ILM approach. To evaluate the full capability of the technique further experiments are required that test the mechanism on different OSs and use the improvements stated within this paper. In spite of that, the current results are promising and we therefore encourage other researchers to explore this ILM method as well as the here proposed concept of PMC-based trapping.

[2]See Chapter 18.4.2 of Volume 3 of the Intel 64 and IA-32 Architectures Software Developer's Manual [8] for details.

6. REFERENCES

[1] S. Bhansali, W.-k. Chen, S. D. Jong, A. Edwards, R. Murray, M. Drinic, D. Mihocka, and J. Chau. Framework for Instruction-level Tracing and Analysis of Program Executions. In *Proc. of VEE*, 2006.

[2] J. Chow, T. Garfinkel, and P. M. Chen. Decoupling dynamic program analysis from execution in virtual environments. In *Proc. of USENIX ATC*, 2008.

[3] A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: Malware Analysis via Hardware Virtualization Extensions. In *Proc. of CCS*, 2008.

[4] J. Du, N. Sehrawat, and W. Zwaenepoel. Performance Profiling of Virtual Machines. *SIGPLAN Not.*, 2011.

[5] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. *SIGOPS Oper. Syst. Rev.*, 2002.

[6] T. Garfinkel, K. Adams, A. Warfield, and J. Franklin. Compatibility is Not Transparency: VMM Detection Myths and Realities. In *Proc. of HotOS*, 2007.

[7] T. Garfinkel and M. Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proc. of NDSS Symposium*, 2003.

[8] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual, 2011.

[9] C. Malone, M. Zahran, and R. Karri. Are hardware performance counters a cost effective way for integrity checking of programs. In *Proc. of STC*, 2011.

[10] A. M. Nguyen, N. Schear, H. Jung, A. Godiyal, S. T. King, and H. D. Nguyen. MAVMM: Lightweight and Purpose Built VMM for Malware Analysis. In *Proc. of ACSAC*, 2009.

[11] R. Nikolaev and G. Back. Perfctr-Xen: A Framework for Performance Counter Virtualization. *SIGPLAN Not.*, 2011.

[12] J. Pfoh, C. Schneider, and C. Eckert. Exploiting the x86 Architecture to Derive Virtual Machine State Information. In *Proc. of Secureware*, 2010.

[13] J. Pfoh, C. Schneider, and C. Eckert. Nitro: Hardware-based System Call Tracing for Virtual Machines. *Adv. in Information and Comp. Sec.*, 2011.

[14] B. Sprunt. The Basics of Performance-Monitoring Hardware. *IEEE Micro*, 2002.

[15] A. Vasudevan and R. Yerraballi. Stealth Breakpoints. In *Proc. of ACSAC*, 2005.

[16] H. Yin and D. Song. TEMU: Binary Code Analysis via Whole-System Layered Annotative Execution. *University of California at Berkeley*, 2010.