Senior Thesis

Raj Ramamurthy
rmmrthy2 @ illinois.edu

December 10, 2015

1 Introduction

This report summarizes my work with the DEPEND research group as part of CS499 (senior thesis) from January 2015 through December 2015.

The paper is structured as follows. A brief discussion of related performance counters work is provided in Section 2. An overview of the userspace data and QEMU is presented in Section 3. In Section 4, the development of the kernel module and interaction with KVM is explained. Section 5 contains an overview of the data collection and architecture. Section 6 analyzes the data that was collected, with areas for future work presented in Section 7. Finally, the conclusions from this work are available in Section 8.

1.1 Timeline

This project took place over two semesters. In the first semester, the initial plan for the work was developed and several core components were engineered. This includes the user space counter collection tool and the kernel space tool.

The second semester of this work was mostly focused around understanding whether performance counters can be a useful invariant on their own, or whether other components would need to be used in order to detect execution irregularities. This involved collecting data on multiple Linux machines and VMs, creating tools to analyze the data, and graphing the data to understand its usefulness. Additionally, most of this semester was spent brainstorming potential applications of the performance counter work and deciding which scenarios this work could be useful in.

1.2 Background

This thesis was grown out of previous semsters I had spent working with the DEPEND research group. Specifically, in this work, I sought to answer the following question: *by constructing fingerprints from execution signatures (using performance counters) of a hypervisor (e.g., qemu-kvm) at different points in time, is it possible to detect compromised hypervisor execution by comparing those signatures to known good ones?* This is a form of *dynamic analysis* because the detection is done at performed by running the hypervisor and not through a tool that analyzes a binary beforehand.

1.3 Hardware Performance Counters

Hardware performance counters (herein referred to as HPCs) allow for the collection of various low-level information about a processor's current state. The counters can provide valuable performance insights. Unfortunately, the first machine I attempted to use did not support HPCs, and it is not enabled by default on most machines. However, after switching to another machine, I was able to enable the counters and begin collecting data.

The counters available vary from machine to machine. The important counters are total retired instructions, conditional branch instructions, branch instructions, store instructions, and load instructions. There are multiple ways to access the counter values. In user space, I used a library called PAPI[2], and in kernel space, I made use of the built-in performance counter API in Linux.

1.4 Motivation

To understand the motivation behind this project, it is important to have a good high-level understanding of the virtualized system topology. A visual overview of the topology is shown in Figure 1.

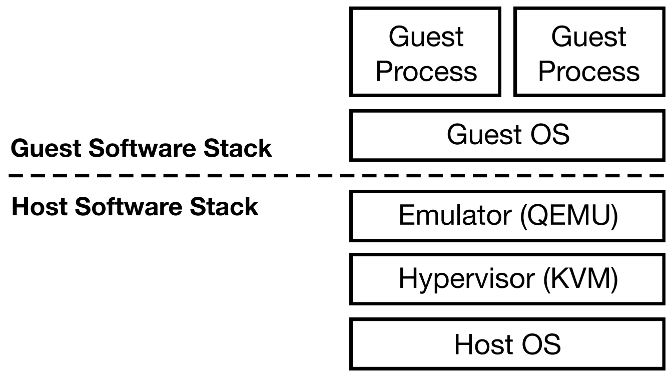


Figure 1: Virtual machine topology. Areas of focus include QEMU and KVM.

A search of the National Vulnerability Database shows that there are over 100 reported vulnerabilities in QEMU. These range in severity, but can be as grave as allowing for arbitrary code execution by the virtual machine guest.[1] While there are research papers already published on dynamic analysis using hardware performance counters, these have not been applied towards virtualization[8][1][5].

The principal motivation for this work was to understand if we could feasibly develop a method of detecting a compromised hypervisor. Hypervisor security is very important, and because the hypervisor functions as the backbone of a virtualized system, its integrity is paramount for the virtual machine's security.

[1]`http://web.nvd.nist.gov/view/vuln/search-results?query=QEMU&search_type=all&cves=on`

2 Related Work

There are a number of existing works which cover HPCs for security. These center around several concerns:

- Performance overhead in using HPCs
- Accuracy and nondeterminism of the counters
- Accuracy of detection methods using counters
- Feasability of HPCs as a dynamic analysis invariant

2.1 Using Counters for Dynamic Analysis

The paper "Are Hardware Performance Counters a Cost Effective Way for Integrity Checking of Programs?" attempts to understand performance counter usage for integrity checking from a cost perspective, and concludes that HPCs are efficient for detecting program modification[4]. The NumChecker[8] paper presents a framework through which to detect kernel rootkits. It uses HPCs to verify system call execution.

2.2 Overhead

As the name implies, hardware performance counters were initially designed to aid in performance instrumentation of low-level software. Using HPCs is attractive because they are built into the processor, meaning that they add a negligible amount of hardware overhead to program execution. However, storing this information and analyzing it dynamically can introduce overhead. The NumChecker[8] paper reveals that the amount of overhead varies significantly with the frequency of collection, presumably due to I/O; for a sampling rate of 5 seconds, the overhead introduced by NumChecker was 2.8%[8]. Another work shows that the overhead incurred in that situation was less than 10%[4].

2.3 Counter Accuracy and Nondeterminism

Using HPCs as an invariant has proved a controversial topic, as they appear to be nondeterministic. This is because there are many events happening on the processor, which cause small fluctuations in the counter values. These include interrupts and other processes running on the system.

However, some metrics are more consistent than others. From [4]: "The shortlisted events include the total number of instructions retired, the number of branch instructions retired, cache stores (they are better measures than loads because loads can be speculative), completed I/O operations, and number of floating point operations."[4]

There are many sources of nondeterminism, and it appears to vary from chip to chip. The paper "Non-determinism and overcount on modern hardware performance counter implementations" explores these sources and attempts to make sense of which counters are best. It comes to the conclusion that total instructions retired is one of the most consistent counters[10]. This paper also discusses the issue of overcount, which occurs in virtually every counter.

2.4 Detection Accuracy from Using HPCs

HPCs only provide the data. From this data, is important to conduct proper statistical analysis to achieve meaningful results. Specifically, in integrity checking, the most important metric is a low false positive rate.

The HPC data is best interpreted as a matrix of feature vectors. From this, a number of very interesting analytical techniques arise, including (but not limited to): linear regression, support vector machines, random forests, nearest neighbors, and clustering. Principally, nearest neighbors is the most popular of these choices, but each approach has different advantages[3]. It is also possible to build simple classification simply by looking within a threshold of all known values[8]. In one paper, artificial neural networks were used[1].

3 QEMU

QEMU is a machine emulator which allows for the virtualization of an operating system. It can function as a software hypervisor and emulate an entire machine. QEMU provides emulation of both entire machines and external I/O devices (e.g., hard-drives, network devices, and keyboards). However, when the host and guest operating system are the same architecture[2], native virtualization can be accomplished by using KVM, a Linux kernel module which allows for hardware-assisted virtualization. KVM leverages processor support for virtualization to expose an API for running guest virtual machines. In this configuration, QEMU only emulates I/O devices. Both pieces of software play crucial roles in running the guest virtual machine, and execution frequently traps from one to another. This is explained in Figure 2.

3.1 Modifications

In order to collect HPC data from QEMU, some modifications were required. QEMU does not have a plugin architecture, so a custom build of QEMU was created for instrumentation purposes.

The modifications were fairly simple: using the PAPI performance counter library, I was able to output HPC measurements across a known path of execution. The path of execution chosen measures the lifespan of an exit in QEMU. That is, the counter is started when the exit comes back to QEMU and ended when the next exit leaves.

Furthermore, with the KVM hypercall extension, I was able to make markings in the data collection from within the guest VM. With this toolset, it

[2]An `x86` machine may also be virtualized on an `x86_64` architecture

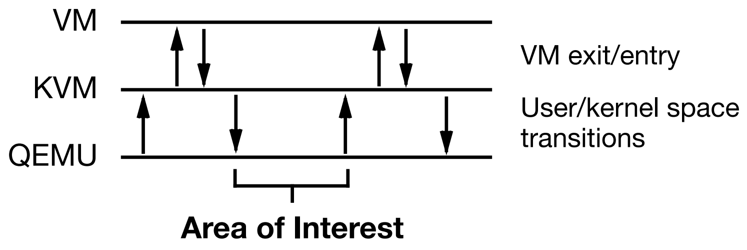


Figure 2: Diagram of the various transfers of control between QEMU, KVM, and the host virtual machine. When KVM returns to QEMU, it provides an exit reason which can be used by QEMU to take appropriate action. However, KVM may transfer control to the virtual machine (and back) before returning to QEMU. The arrows represent transfer of control. The duration for which HPCs were recorded is also displayed. This area was chosen because it focuses solely on QEMU while allowing for the collection of the exit reason for KVM, allowing us to only collect counter data from the QEMU binary.

became trivial to mark the beginning and ending of a program in the guest VM and observe the QEMU data for this period.

4 KVM and `kprof`

The KVM kernel module was modified for use in this project. This section aims to explain the changes that were made and elaborate on the design of the additional data collection module that was developed.[3]

4.1 KVM Exits

When execution is transferred to KVM, the hypervisor will return what is known as an *exit reason* when execution transfers back. If the exit was generated by VT-x (a privileged instruction was executed and the VM is using hardware-assisted virtualization, KVM will label the exit appropriately corresponding to its context. In other exits not generated by VT-x, KVM will also place a label on the exit reason. There is a long list of exit reasons spanning many different explanations for what occurred while KVM had control of execution. [4]

[3]A full list of modifications to the module is available in the README of the project.

[4]A full list of the exit reasons for KVM is available at `http://lxr.free-electrons.com/source/arch/x86/include/uapi/asm/vmx.h#L30`

4.2 The `kprof` Module

In order to be able to collect the data from user space, I developed a device driver. This allows the usage of the `ioctl` function from the user space program (useragent) and for data to be transferred out of the kernel.

In the device driver, called `kprof`, a set of internal buffers in constructed so that the vectors of performance counter data may be continuously populated in memory and then flushed to disk at an interval (this is the same technique used in the hprobe kernel module). This helps keep the performance impact of this double buffer solution manageable in addition to providing a lock-free mechanism for reading the data from multiple code paths.

The current kprof API has the following functions:

1. `start_record()`, which starts the recording of events.
2. `stop_record(exit_reason)`, which stops the recoding, records the difference in the counter values since the last call to `start_record`, and records the exit reason given for this data entry.

I then modified the KVM kernel module to make these calls in the exit handler. Specifically, the counter values recorded represent execution across the KVM code path. When an exit returns, the counters are started until the next exit comes in.

As discussed above, the counters are placed into a series of buffers. Finally, the useragent program is run in user space to flush these buffers and reset the internal state of the `kprof` module. Because a single run of the VM may result in the collection of millions of entries (resulting in some buffers being overwritten), the useragent program is run simultaneously while the VM is running. One major benefit of this multi-part solution is that the module does not need to be re-installed for additional runs. An API for resetting the internal state of the module is exposed via the `ioctl` interface.

5 Data Collection

Data for this project was collected using both the `kprof` kernel module and the userspace QEMU extensions. The data was collected from runs of a CentOS 7 virtual machine when using Linux. For Windows, the VM was a Windows 7 VM. Details of the userspace data collection are available in [6]. An overview of the entire system is shown in Figure 3.

6 Analysis

6.1 iozone and Workload Distribution

Figure 4 shows a plot of unique counter values observed over time for a workload consisting of machine boot followed by several runs of the iozone benchmark.

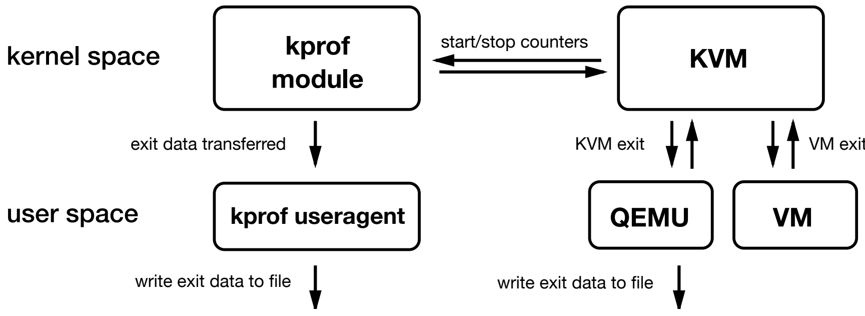


Figure 3: System architecture overview composing both userspace and kernel space data collection mechanisms. Two unique files are generated. Each line in the file represents a row of tab-separated data: the first entry is the exit reason, which is followed by the performance counter values.

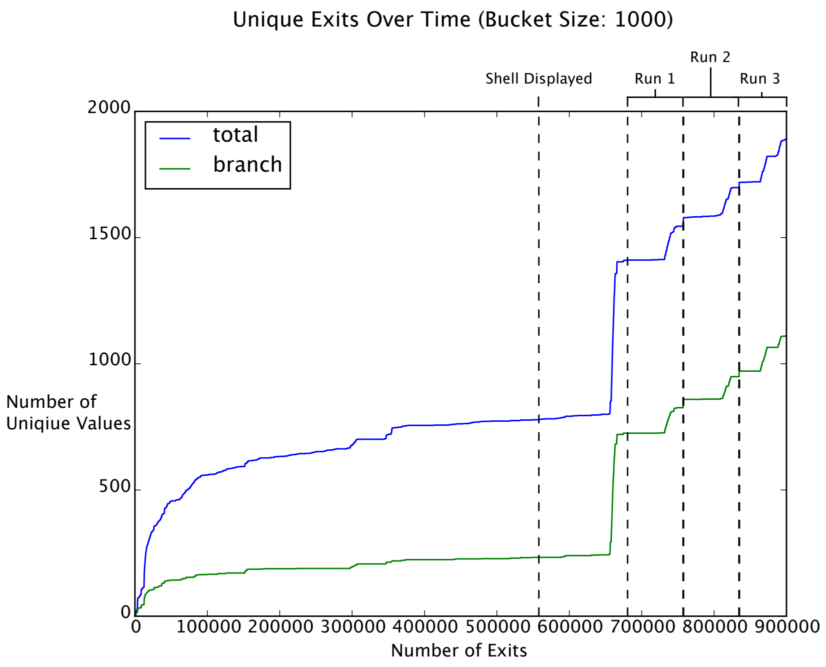


Figure 4: Plot of the unique counter values observed over time. The workload consists of initial boot of a Linux machine, followed by three succesive runs of the iozone filesystem benchmark. Here, *bucket size* refers to a smoothing factor applied to the data. Numbers are placed into buckets of this size. A bucket size of 1000 means there are buckets of values 0-999, 1000-1999, etc. Numbers in the same bucket are treated as the same for the creation of this graph.

Multiple data collection trials and analysis showed that the counter data is consistent across trials. A visual evaluation of the data as presented in Figure 4 shows that we cannot make an accurate judgement of whether we are seeing a new wokload purely based on the performance counter data. From this graph, we observe that the workload manifests as a change in the graph; after the workload starts, more unique counter values are observed. Additionally, this graph gives some insights into how the operating system uses the hypervisor. We observe a high level of entropy at boot and shortly after the shell is displayed to the user. Runs of the filesystem benchmark increase the number of unique counters observed, but successive runs do not show an increase by a lesser delta. From this, we can conclude that while the performance counters are a valuable metric, they cannot stand alone in detecting certain executions.

This is just one view of the data. Another is to separate the graphs by observed exit reason. This allows us to interpret the specific nature of the workload. For the iozone dataset, if we separate the data according to exit reason, there are many `HALT` exit reasons occuring during the iozone runs (this exit reason indicates that the CPU has been paused until the next interrupt occurs). However, these are sparse during the boot of the operating system. In this case, the halt instructions might be a sign that there is an IO-heavy operation occurring or some other device-related operation. These sorts of observations are interesting for understanding more about the workload.

6.2 Windows and Linux

One of the interesting applications of the counter data is to better understand how Linux and Windows use the hypervisor. We can do this by collecting a sample of counter values through the kprof module from booting a virtual machine of both systems.

I collected this data from the aforementioned CentOS VM and a Windows 7 VM, both running on the same bare metal machine with the same hypervisor and custom QEMU fork. [5] The data was then fed through a series of scripts to split it out by exit reason and observe how the number of unique exit reasons grow over time. This script used a bucket size of 1000 (refer to Figure 4 for more information on bucket size).

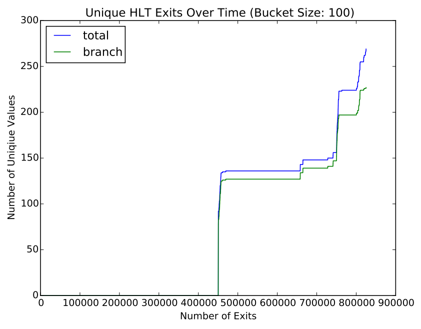
This is a black-box style approach because the hypervisor is not aware of what is running above it in the stack. From the area in code where the performance counter data gets collected, it is not possible to be entirely sure what code caused the exit to occur or what operating system is being run. However, by analyzing trends in the counter data that is collected, we can make an educated guess as to which operating system is being run. This is because Windows and Linux use the hypervisor in different ways over time.

The two systems have a different distribution of total exit reasons, with Windows having more EPT_MISCONFIG and PENDING_INTERRUPT exits

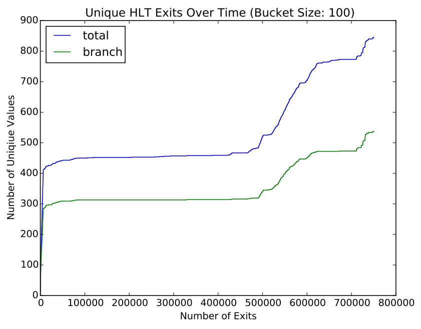
[5]Note: on boot, the Linux VM is able to make a hypercall so as to exactly mark when startup is complete, but this is not possible with the Windows machine.

than Linux. Both boot sequences have high levels of IO_INSTRUCTION exits compared other reasons. There are also some differences in how the exits occur over time. It is difficult to come to any specific conclusion from this exit reason distirbution, but it may be helpful in aiding future tools to detect which system is being run.

Ultimately, the data presented in this section is useful because it provides insights into how these two operating systems leverage the hypervisor. Looking at the data without exit reasons, it's clear that the trends are different. However, it is even more powerful to look at the data based on each exit reason. In this manner, this data can expose how different subsystems of the operating system leverage the underlying hardware.



Windows



Linux

Figure 5: Distribution of HLT (halt) exits for Windows and Linux virtual machine boots. Observe that the number of HLT exits quickly climbs and stabilizes for Linux, but does so much later in the Windows process. It is difficult to explain this discrepancy, but it does give some insight that we can use the counter instrumentation tools to observe differences in how these two operating systems use the hypervisor when running in a virtual machine. **Note:** these two graphs have different axis. That is because they represent different workloads.

6.3 KVM and QEMU

The next area of interest is lookng at how both the KVM and QEMU measurements portray the same workload. In Figure 6, a boot sequence of the CentOS VM has been graphed using both subsystems' counters. This graph shows us that KVM generates a significant amount of data compared to QEMU, but also that growth of KVM counters is more granular. In order to better understand the repetitive nature of these datasets, I hashed each row of data to check what percentage of the data was actually unique. The results were surprising: only 2.1% of the QEMU signaures were unique compared to 30.74% of the KVM signatures. Therefore, even though KVM sees many more exits than QEMU does, the performance counter signature (the values combined with the label) measured across the exit is more likely to have been seen before. This suggests that there is more entropy in the QEMU subsystem.[6]

6.4 Feasability of Whitelist Construction

Because the number of unique counters never truly reaches saturation, it is unfortunately not possible to rely solely on hardware performance counters to determine whether a unique (or safe) workload is being run. It may, however, be possible to combine data from other areas of the stack with these HPCs to build a new counter value. One possible solution is to look at the basic blocks of instruction and attempt to infer unique workloads from the control flow of the machine.[7]

7 Future Work

Future work will involve building tools around the workload analysis component of this work. Since the API is already there, this is not too difficult to do for other developers. They can easily grab the counter data and manipulate it. I envision many interesting tools stemming from this. One example is a quick tool to capture the current hypervisor state and graph the next 30 seconds of data to see if there are any interesting phenomena occurring.

The other potential application is to continue working with this data in order to build a new type of counter (as outlined in the previous section). Further research is required to understand and determine what other sources of data would be good candidates for building such a counter.

Additionally, the Air Force Research Laboratory has expressed interest in this work and may decide to use it in other projects.

7.1 Open Questions

The primary open question is: how can we leverage this existing performance counter infrastructure and data to create security tools? . From there, a hardware unit can be built to continuously monitor integrity. Since the overhead of the performance counters is relatively minimal (due to native support in the CPU), this could be a viable application of this work.

Since this is a black box approach, another question is what more we can learn about the operating system (and other potential workloads running within it) from this data alone. I have shown that it is not possible to uniquely isolate a workload, but some other insights about the characteristics of any general computation being done using the hypervisor could be drawn from the data.

[6]The data was collected with interrupts pinned to CPU 0 and the QEMU process pinned to CPU 1.

[7]This approach was originally outlined in the hShield paper earlier in the year.

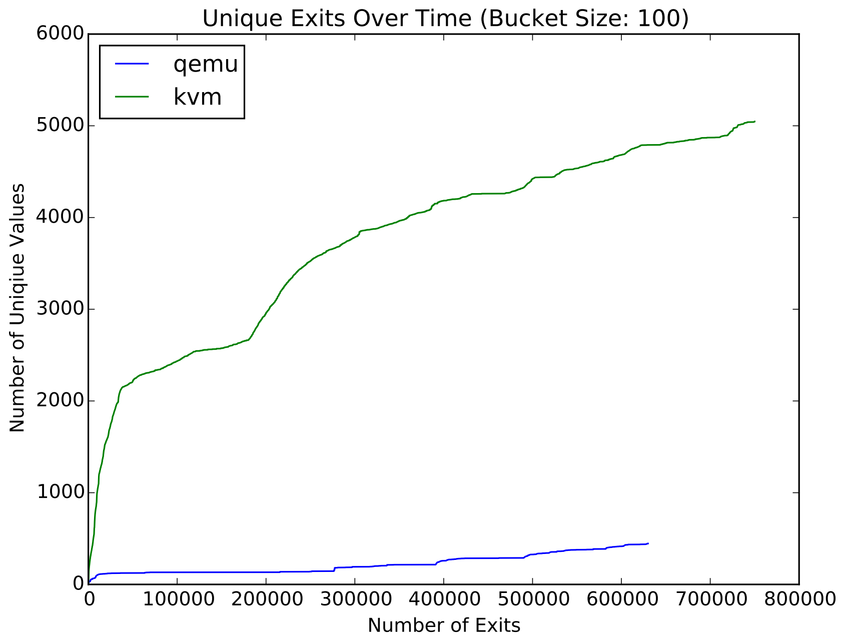


Figure 6: Unique QEMU and KVM exits over time. This is based on the total instructions performance counter. The QEMU data was collected with PAPI. The KVM data was collected using the kprof kernel module. The general upward trend of seeing unique exits is true for both components across the boot, but KVM sees far more exits. Upon adjusting the bucket size it becomes very clear that KVM is seeing far more "clustered" exit values than QEMU does. Since these are collected from the same workload, it is interesting to see that there are some discrepancies.

8 Conclusion

This work has shown that hardware performance counters can provide valuable infomration about how the hypervisor is used. It has also shown that HPCs alone cannot be used as an invariant for dynamic security analysis, and it has helped motivate the future direction of the hshield work.

One very valuable product of this work is the creation of the *kprof* module, which allows any user-space process (or kernel module) to easily and efficiently capture and extract performance counter data. My hope is that this module is further developed into a powerful suite of analysis tools that any program can quickly invoke.

Personally, I have learned many interesting facts about how virutal machines operate and how other subsystems of Linux work. Specifically, I've learned the effects of interrupts on program execution, how to use performance counters to instrument execution, and how to develop an efficient device driver.

References

[1] John Demme, Matthew Maycock, Jared Schmitz, Adrian Tang, Adam Waksman, Simha Sethumadhavan, and Salvatore Stolfo. On the feasibility of online malware detection with performance counters. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 559–570, New York, NY, USA, 2013. ACM.

[2] Jack Dongarra, Kevin London, Shirley Moore, Phil Mucci, and Dan Terpstra. Using PAPI for hardware performance monitoring on Linux systems.

[3] David A. Forsyth. *Probability and Statistics for Computer Scientists.* 2014.

[4] Corey Malone, Mohamed Zahran, and Ramesh Karri. Are hardware performance counters a cost effective way for integrity checking of programs. In *Proceedings of the Sixth ACM Workshop on Scalable Trusted Computing*, STC '11, pages 71–76, New York, NY, USA, 2011. ACM.

[5] David Molnar, Matt Piotrowski, David Schultz, and David Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In *Proceedings of the 8th International Conference on Information Security and Cryptology*, ICISC'05, pages 156–168, Berlin, Heidelberg, 2006. Springer-Verlag.

[6] Raj Ramamurthy. Independent study research report. `http://web.engr.illinois.edu/~rmmrthy2/papers/fa14-report.pdf`, Dec 2014.

[7] Raj Ramamurthy, Zachary Estrada, Cuong Pham, Zbigniew Kalbarczyk, and Ravishankar Iyer. Designing a performance isolation benchmark for virtualized systems. In *The 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks.* DSN, 2014.

[8] Xueyang Wang and R. Karri. Numchecker: Detecting kernel control-flow modifying rootkits by using hardware performance counters. In *Design Automation Conference (DAC), 2013 50th ACM / EDAC / IEEE*, pages 1–7, May 2013.

[9] Xueyang Wang and R. Karri. Numchecker: Detecting kernel control-flow modifying rootkits by using hardware performance counters. In *Design Automation Conference (DAC), 2013 50th ACM / EDAC / IEEE*, pages 1–7, May 2013.

[10] Vincent M. Weaver, Dan Terpstra, and Shirley Moore. Non-determinism and overcount on modern hardware performance counter implementations. *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 0:215–224, 2013.

[11] V.M. Weaver and S.A. McKee. Can hardware performance counters be trusted? In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pages 141–150, Sept 2008.

[12] Matas Zabaljuregui. Hardware assisted virtualization intel virtualization technology. `http://linux.linti.unlp.edu.ar/images/f/f1/Vtx.pdf`, June 2008.

[13] D. Zaparanuks, M. Jovic, and M. Hauswirth. Accuracy of performance counter measurements. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 23–32, April 2009.

[14] Xiao Zhang, Sandhya Dwarkadas, Girts Folkmanis, and Kai Shen. Processor hardware counter statistics as a first-class system resource. In *Proceedings of the 11th USENIX Workshop on Hot Topics in Operating Systems*, HOTOS'07, pages 14:1–14:6, Berkeley, CA, USA, 2007. USENIX Association.