On the Detection of Kernel-Level Rootkits Using Hardware Performance Counters

Baljit Singh
Department of Computer Science and Engineering
Qatar University
baljit92@gmail.com

Dmitry Evtyushkin
Department of Computer Science
State University of New York at Binghamton
devtyushkin@cs.binghamton.edu

Jesse Elwell
Vencore Labs
jelwell@vencorelabs.com

Ryan Riley
Department of Computer Science and Engineering
Qatar University
ryan.riley@qu.edu.qa

Iliano Cervesato
Computer Science Department
Carnegie Mellon University
iliano@cmu.edu

ABSTRACT

Recent work has investigated the use of hardware performance counters (HPCs) for the detection of malware running on a system. These works gather traces of HPCs for a variety of applications (both malicious and non-malicious) and then apply machine learning to train a detector to distinguish between benign applications and malware. In this work, we provide a more comprehensive analysis of the applicability of using machine learning and HPCs for a specific subset of malware: kernel rootkits.

We design five synthetic rootkits, each providing a single piece of rootkit functionality, and execute each while collecting HPC traces of its impact on a specific benchmark application. We then apply machine learning feature selection techniques in order to determine the most relevant HPCs for the detection of these rootkits. We identify 16 HPCs that are useful for the detection of hooking based roots, and also find that rootkits employing direct kernel object manipulation (DKOM) do not significantly impact HPCs. We then use these synthetic rootkit traces to train a detection system capable of detecting new rootkits it has not seen previously with an accuracy of over 99%. Our results indicate that HPCs have the potential to be an effective tool for rootkit detection, even against new rootkits not previously seen by the detector.

Keywords

Rootkits; Hardware Performance Counters; Intrusion Detection; Machine Learning

ASIA CCS '17, April 4–6, 2017, Abu Dhabi, United Arab Emirates.

© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-4944-4/17/04…$15.00.

DOI: http://dx.doi.org/10.1145/3052973.3052999

1. INTRODUCTION

Recently, work has been done investigating the use of hardware performance counters (HPCs) for the detection of malware [6, 18, 27, 29]. The goal of these works is to detect malware based on a profile of the way it impacts performance counters that are included in the processor of the machine. This is a form of behavioral detection. Existing work has focused on running malware binaries while collecting performance counter information and using that data to train a malware detector using various machine learning techniques. Initial results have been very promising, demonstrating detection rates of over 90%.

One type of malware detection that has thus far not shown promising results is the detection of kernel rootkits. Rootkits are a special type of malware that modifies parts of the running operating system kernel in order to hide the presence of an attacker on a machine. There are a variety of attack methodologies a rootkit might use such as code-injection, direct kernel object manipulation, function pointer hooking, and more. One thing that makes rootkit analysis unique when compared to traditional malware is that the rootkit's functionality does not execute in its own process context, instead the functionality executes in the context of *other* processes that access kernel information. In [6] a preliminary set of tests were performed to detect rootkits, but the results were not encouraging and were not investigated deeply.

In this work, we aim to provide a more comprehensive analysis of the applicability of hardware performance counters to the detection of kernel rootkits. We experimentally demonstrate how various types of rootkit functionality and attack mechanisms impact HPCs and determine the most significant HPCs for use in detecting rootkits. Our results indicate that the HPCs are most impacted by the mechanism of attack (function pointer hooking, system call hooking, etc.) and less impacted by the rootkit functionality (file hiding, process hiding, etc.)

We then design, train, and test a machine learning based rootkit detector capable of detecting rootkits attacks against a Windows 7 computer. Our results indicate that a system trained on a variety of rootkit attack mechanisms can detect new rootkits that use those same mechanisms, even if they are not variants of each other.

The contributions of this work are as follows:

- We provide an evaluation of the impact on HPCs of five different types of rootkits employing the three most common attack techniques. We identify 16 HPCs (from over 400) that are the most significant for rootkit detection.

- We find that one class of rootkits, those employing direct kernel object manipulation (DKOM), do not have a significant impact on HPCs and thus cannot be detected by this technique.

- We design, train, and test a machine learning based rootkit detection system capable of detecting rootkits. This demonstrates the efficacy of HPCs for the detection of hooking based rootkits.

- Our detector is able to detect previously unseen rootkits based on their attack mechanisms, implying that HPCs can be used to detect zero-day rootkit attacks as long as those attacks employ known attack mechanisms.

- We discuss the practical limitations of using HPCs for rootkit detection and provide recommendations for hardware modifications that would address these limitations.

2. BACKGROUND

In this section we will present a brief background on hardware performance counters, rootkits and the use of hardware performance counters for malware detection.

2.1 Hardware Performance Counters

Performance monitoring [26] is an essential feature of a microprocessor. Access to the performance monitoring hardware is usually provided in the form of hardware performance counters (HPCs), a collection of configurable, special-purpose registers in recent microprocessors. Such counters can be found in many microarchitectures. Today, all of the major processor platforms have support for HPCs [2]. These counters are used to obtain low-level information on events happening in the hardware during program execution. HPCs are most often utilized in order to find bottlenecks in critical parts of programs, for fine-grained application tuning, compiler optimizations, or to study peculiarities of program behavior on various CPUs. HPCs are capable of counting events associated with many types of hardware-related activities such as clock cycles, cache hits/misses, branch behavior, memory resource access patterns and pipeline stalls, etc.

Each HPC register can be configured to count events of a particular type. After the configuration, each time the hardware event detector detects a specific event, the counter will be incremented. Access to the counter registers is performed using special purpose instructions. Usage of HPCs is beneficial for program behavior analysis, since they offer very high accuracy [31] and normally do not introduce slowdown to program execution. HPCs are also used for other purposes; for example, they can be used for power [25] and temperature [12] analysis. Another beneficial use is monitoring program behavior for malware detection [6, 18, 27, 29] and integrity checking [14].

Despite the large number of possible events that HPC registers can be configured to count, there is a limitation common to all platforms: the limited number of configurable registers. For example, the Intel Ivy-bridge and Intel Broadwell CPUs used in this work can be configured to capture 468 and 519 events respectively, but the number of counter registers is limited to only four per processor core, meaning that only four HPCs can be captured simultaneously. This limitation can be mitigated by multiplexing performance counters [16], but at the cost of accuracy. Finally, there are many libraries and software toolkits available for accessing HPCs, such as [3, 5] and [13].

2.2 Kernel Rootkits

Kernel rootkits (referred to as simply rootkits in this paper) are a type of malware that modifies the running OS kernel with the intention of hiding the malware's presence on a system. Frequently a rootkit author wants to hide a running process, conceal an installed driver, mask the existence of a file on the file system, hide incoming and outgoing network connections, etc. The methodology used by rootkits to accomplish these goals varies as well. They can use system call table hooking [9], function pointer hijacking [30], direct kernel object manipulation (DKOM) [9], and more. Even the ways in which they execute their malicious logic can vary from kernel-level code injection to return-oriented programming [10] to not executing code in the kernel at all [21, 22].

2.3 Hardware Performance Counters for Intrusion Detection

Recently, work has been done applying HPCs to intrusion detection. In this section we discuss the piece of seminal work in the area, but a more complete handling of related work can be found in Section 6.

Work by Demme et al. [6] shows the feasibility of using HPCs to detect malware. They used micro-architectural features from ARM and Intel processors to successfully detect malware on Android. Their approach involves capturing multi-dimensional, time-series traces of running applications by interrupting periodically and capturing all performance counters for the current thread of control. After training on existing malware, they were able to detect variants with high accuracy: Over 90% with a false positive rate (FPR) less than 10%. In addition to their Android based results, they provided a brief set of experiments attempting to detect rootkits on Linux, but the results were not nearly as promising: Around 70% accuracy at a 10% FPR. In commenting about their rootkit results, they say the following: "... we believe our rootkit detection shows promise but will require more advanced classification schemes and better labeling of the data to identify the precise dynamic sections of execution that are affected."

Motivated by their experiments and hypothesis, in our work we provide a more comprehensive set of experiments spanning a variety of rootkit types and demonstrate that HPCs can be significantly more effective for rootkit detection that their initial results indicate.

Table 1: Synthetic Rootkits Used for Testing

Name	Functionality	Attack Mechanism
SR1	Hides targeted outgoing TCP connections and prevents them from being visible to applications like netstat	IRP Hooking
SR2	Hides specific files by preventing them from appearing in any file listings or file managers	IRP Hooking
SR3	Hides processes, preventing them from being listed in process listings	SSDT Hooking
SR4	Hides specific files by preventing them from appearing in any file listings or file managers	SSDT Hooking
SR5	Hides processes, preventing them from being listed in process listings	DKOM

3. LINKING PERFORMANCE COUNTERS TO ROOTKIT FUNCTIONALITY

Given that a modern processor has access to over 400 performance counters, a prudent first step toward detecting rootkits using HPCs is to determine which of those 400 are most significantly impacted by rootkits. In this section we describe a set of experiments designed to determine which HPCs are most impacted by rootkits.

An overall diagram of our approach can be found in Fig. 1. We start with a set of custom designed, synthetic rootkits. These are rootkits that we created that each implement a single piece of rootkit functionality using one attack mechanism. From there, a rootkit is chosen and installed on a Windows 7 virtual machine. Inside that virtual machine a profiling benchmark is executed. This benchmark is a program designed to make use of various pieces of OS functionality that a rootkit will typically impact. (For example, showing a listing of all running processes.) While the benchmark is running we use Intel's VTune [11] to capture traces of all the possible HPCs during the execution of the benchmark. We then process our traces using the Gain Ratio feature selection technique from the WEKA machine learning toolkit[8] to determine which features are the most significant for each synthetic rootkit.

3.1 Synthetic Rootkits

Synthetic rootkits are small rootkits which are designed to make use of a single attack mechanism to accomplish a specific rootkit goal. Most real-world rootkits make use of multiple attack mechanisms and exhibit more than one type of functionality. By making use of specialized, synthetic rootkits, we are able to more precisely link impacted performance counters with the specific attack mechanism or functionality that causes the impact.

Our synthetic rootkits focus on three major attack mechanisms that are found in rootkits on the Windows 7 platform: I/O Request Packet (IRP) Hooking, System Service Dispatch Table (SSDT) Hooking and Direct Kernel Object Manipulation (DKOM). Table 1 gives a brief summary of the five synthetic rootkits used in this work and each of them is described in more detail below.

SR1: Network port filtering using IRP Hooking SR1 makes use of a technique called I/O Request Packet (IRP) hooking, which is a type of function pointer hijacking. Each device in Windows is represented as a device object in the OS managed by the I/O Manager. Whenever communication is supposed to take place between the device and an application, an I/O request packet is created and it passes through an abstraction layer. The abstraction layer consists of several drivers that each perform different functions (for example, the disk driver deals with disk read/write requests). For each driver, there are several major functions that are called whenever an IRP passes through the driver. These functions are listed in a table of function pointers.

When a rootkit performs IRP hooking, it replaces one or more of these function pointers with a pointer to a custom-built, malicious version of the function that has been loaded by the rootkit into OS memory (in a loadable, kernel-level driver, for example). The malicious versions of these functions can then modify the contents of the IRP or further divert control-flow as required.

Using this technique, we have hooked a function pointer in the network driver in order to filter out all the outgoing network connections on port 80. To accomplish this, we needed to hook within the driver loaded from `TCPIP.SYS` and filter the list of active network connections before it is passed from the kernel to the application layer (where it is ultimately displayed to the user). The driver object, `\\DEVICE\\TCP`, further points to a table containing the major IRPs. The IRP we focus is `IRP_MJ_DEVICE_CONTROL`, which originally calls an IRP handler (the default IRP handler). The IRP handler returns the requested network data.

In `IRP_MJ_DEVICE_CONTROL`, we look at the `IOCTL_TCP_QUERY` control code which returns the list of network ports currently in use to `netstat.exe`.

To filter the results, `IRP_MJ_DEVICE_CONTROL` is shifted to point to our custom-designed IRP handler. The custom IRP handler further calls the default handler that returns the required data and fills the output buffer. Once the required data returns, we can process the data in the output buffer according to our needs. To hide outgoing connections on port 80 all that needs to be done is to change the status value of each object in the buffer related to port 80 to `0`. After the parsing is done, we send it to the requester. Fig. 2 illustrates IRP hooking.

SR2: File Hiding using IRP Hooking Our next synthetic rootkit, SR2, uses the same attack mechanism as SR1, but in this case it is used to hide a file rather than a network connection.

A file system filter driver can filter I/O operations for one or more file systems or file system volumes. Depending on the nature of the driver, filter can mean log, observe, modify, or even prevent. Typical applications for file system filter drivers include anti-virus utilities, encryption programs, and hierarchical storage management systems [17].

In this technique, we obtain a handler on the IRP function `IRP_MJ_CREATE`. This function helps us with retrieving the

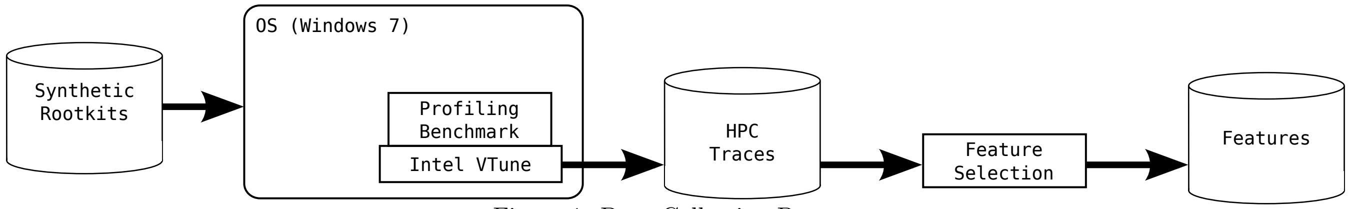
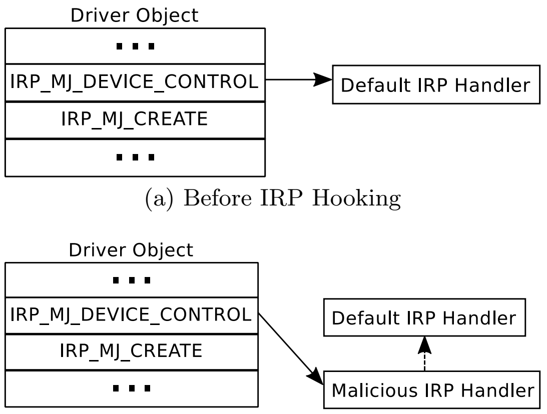
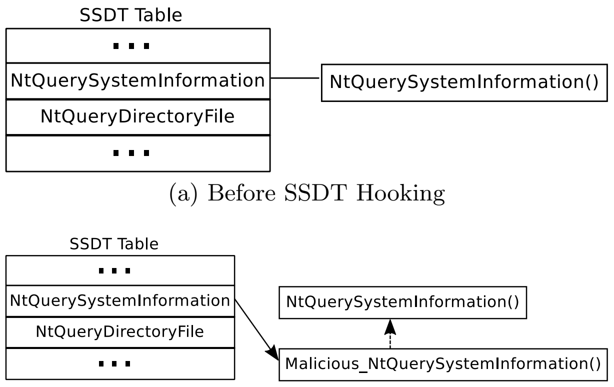


Figure 1: Data Collection Process



(a) Before IRP Hooking

(b) After IRP Hooking
Figure 2: Visualizing IRP Hooking


SSDT Table
...
NtQuerySystemInformation
NtQueryDirectoryFile
...
NtQuerySystemInformation()
Malicious_NtQuerySystemInformation()
(a) Before SSDT Hooking

(b) After SSDT Hooking
Figure 3: Visualizing SSDT Hooking

name of the files as soon as they are opened. Every time `IRP_MJ_CREATE` returns it sends back all the details of the file opened to our custom-function. Once we have the file name we can check the extension of the file and not let it proceed. In our case, we hide files with the `xml` extension under the Windows directory.

SR3: Process Hiding using SSDT Hooking While IRP hooking involves modifying function pointers in various drivers, another location in Windows that can be hooked is the System Service Dispatch Table (SSDT), which is the Windows equivalent of the Linux system call table. This table consists of pointers to service functions exposed to `ntoskrnl.exe`. This table is accessed whenever a system service is requested.

Hooking a function pointer in the SSDT allows an attacker to effectively replace any of the OS system calls. Fig. 3 illustrates this.

We applied this technique in SR3 for the purpose of hiding processes. We hook the service function `NtQuerySystemInformation()` with a malicious version that calls the original version in order to retrieve the list of running processes and then filters it prior to returning the results back to the user space application that made the request. `NtQuerySystemInformation()` returns the list of processes as a linked list. To filter the given process (by name), the process is disconnected from the linked list. Finally, the new, filtered linked list is returned.

Our aim using this technique is that the processes selected by the rootkit to be hidden should be invisible to applications such as the Task Manager or Process Explorer. In addition, they should not be available for Windows API functions and other process APIs.

SR4: File Hiding using SSDT Hooking Our SR4 rootkit applies SSDT hooking for the purpose of hiding files. We hook the service function called `NtQueryDirectoryFile()`. Whenever the above mentioned function is called our malicious version is executed instead of the original function. The way files are hidden is very similar to the way processes are hidden in SR3. When the `NtQueryDirectoryFile()` routine is called it returns a structure array that represents a file. The two fields required are the `FileName` and `NextFileOffset`. To hide a file by its name, the `NextFileOffset` of the current file is set to the `NextFileOffset` address of the file structure to be removed. Similar to SR3, the filtered structure array is then returned to user space.

Specifically, SR4 hides files with names starting with `com`.

SR5: Direct Kernel Object Manipulation Direct Kernel Object Manipulation (DKOM) is a rootkit technique that involves hiding things without the need to hook function pointers or execute injected code. This is very different from the IRP and SSDT hooking attacks described thus far. In order to accomplish this, the rootkit directly modifies OS data structures in memory in order to remove references to items that the user intends to hide.

SR5 uses DKOM to hide processes. Under Windows, every active process is associated with a struct `EPROCESS` in kernel memory. This struct consists of a ListEntry (a linked list) variable with the name `ActiveProcessLink`. This ListEntry further consists of two entries, `FLINK` and `BLINK`. The `FLINK` member of this struct points to the next entry (process) in the doubly-linked list while the `BLINK` member points to the previous entry (process). In order to hide a specific process with a given PID, all we have to do is disconnect it from the doubly-linked list. To do this, we set the `FLINK` of the process preceding the process we want to hide to the `FLINK` of the process we are hiding. The same is done with the `BLINK` of the next process, which is set to the `BLINK` of the process being hidden. This involves the direct modification of kernel memory, and the rootkit can completely unload itself after those modifications occur.

Table 2: Trace Background Workload Conditions

Name	Description
Quiet	The profiling benchmark is executed and there are no background processes running (except those required by Windows).
Noisy 1	The profiling benchmark is executed and there are two background processes running: Downloading a large file over HTTP and listing of the Windows directory in a loop.

3.2 Profiling Benchmark

One unique aspect of rootkits when compared to other types of malware is that after they are installed, they do not execute in the context of their own process. Rootkits that use hooking, for example, have their code executed in the context of whatever process requested the relevant file or process information from the OS kernel. This means that in order to properly collect the HPC data for a rootkit, we cannot profile the rootkit itself. Instead, we must profile another application that causes the rootkit functionality to be triggered.

In this work we construct a profiling benchmark that collects data from the OS that a rootkit might like to hide. The benchmark calls a variety of system programs to gather potentially hidden information. The system programs called by the benchmark are: `netstat`, `ping`, `tasklist`, `open`, `taskkill`, and `dir`.

The benchmark is a continuous loop, but we limit its runtime in our tests to be about 45 seconds.

3.3 Testing Platform

Our test platform (where rootkit infection is performed) is a Windows 7 virtual machine running on VMWare Workstation version 10. We chose VMWare because it allows for easy rollback of the OS after infection (which allows us to easily repeat experiments) and because it has support for virtualizing HPCs. All of the security measures on the system were manually disabled.

In order to capture the HPC traces of the profiling benchmark, Intel VTune 2015 [11] was used. VTune allows applications to be run while capturing a configurable set of HPCs.

In order to allow us to collect many traces quickly, we ran our system on multiple computers with different CPUs: Both Intel Ivy Bridge and Intel Broadwell. Both types of CPUs have full support for HPCs.

3.4 Gathering HPC Traces

For each of our synthetic rootkits, we infected the system with the synthetic rootkit and then used VTune to capture the HPC traces of the execution of our profiling benchmark.

A trace is made up of the final HPC values captured after the entire, 45 second run of the profiling benchmark. This means the traces are not a time series, and no sampling is performed. The raw value of each HPC at the end of the 45 second run is used. Even though every run of the benchmark is fixed at 45 seconds, minor variations in the number of clock cycles per run were observed due to the other activity on the system at the time. In order to correct for this, traces were scaled (through simple division) to ensure that all traces are normalized to a fixed number of clock cycles.

As mentioned previously, there are over 400 HPCs available to be captured. In order to reduce this number, we did some initial rootkit profiling using our benchmark and captured all the HPCs supported by VTune and the hardware. A number of the HPCs were zero for all of these initial tests, and so we removed them from our list, after which 244 HPCs remained.

Due to limitations in the hardware, only four different HPCs can be reliably captured simultaneously. This means that in order to capture data for all 244 HPCs, the benchmark needs to be repeated 61 times, with each run capturing 4 different HPCs. We then combine all of the HPC data to produce one trace with all 244 HPCs.

In order to ensure a variety of background workloads during trace capture, we made use of the two different background workload conditions listed in Table 2. Whenever a trace was captured, one of these two conditions was true. For each testing condition and rootkit combination, 50 traces were collected for a total of 500 infected traces. Given that the profiling benchmark requires 45 seconds per execution, and 61 executions are required to capture all 244 HPCs, it takes about 45 minutes to capture one trace. To capture all 500 traces requires a little over two weeks on a single machine.

In addition, the same procedure is then repeated to gather traces from a clean system which is not infected by any of the rootkits. For the clean system we collect 300 additional traces under each of the two testing conditions. This gives us a total of 1100 traces.

3.5 Most Significant HPCs

With the 500 traces collected from the five synthetic rootkits and 600 clean traces, the next step is to determine which of the HPCs are most significant for detecting each type of rootkit.

For each synthetic rootkit, we determine the most significant HPCs using WEKA [8]. To select the attributes, we use the Gain Ratio Attribute Evaluation Algorithm with full-training set as the selection mode. Once we have the results of the algorithm for each synthetic rootkit, we select only the most significant HPCs (HPCs with confidence level of 1).

Table 3 summarizes the results of our experiments. In total, we identify 16 unique HPCs that are the most significant with respect to the synthetic rootkits. There are a number of observations that can be made from these results.

Synthetic rootkits that use the same mechanism have significant overlap in terms of the most significant HPCs. SR1 and SR2 (IRP hooking) share 2/3 HPCs, while SR3 and SR4 (SSDT hooking) share 9/12 HPCs. This is in contrast to the sharing seen between rootkits with the same functionality but different mechanisms. SR2 and and SR4 both perform file hiding, but do not share any of the most significant HPCs.

Also of interest, but not surprising, is that SR5, the DKOM based rootkit, did not have any HPCs appear as significant. A DKOM based attack directly manipulates the kernel data structures beforehand, meaning that no rootkit code executes during the run of the profiling benchmark. This is a strong indication that HPCs are not a suitable method for detecting DKOM based attacks, and highlights a limitation of using this approach for rootkit detection.

In order to visualize the traces, we applied Principal Component Analysis (PCA) to our synthetic rootkit traces in or-

Table 3: Most Significant HPCs For Synthetic Rootkits

Name	Mechanism	Most Significant HPCs
SR1: Network Port Filter	IRP Hooking	BR_INST_RETIRED.NEAR_TAKEN BR_INST_RETIRED.NOT_TAKEN BR_MISP_EXEC.ALL_BRANCHES
SR2: File Hiding	IRP Hooking	BR_INST_RETIRED.NEAR_TAKEN BR_INST_RETIRED.NOT_TAKEN
SR3: Process Hiding	SSDT Hooking	ICACHE.IFETCH_STALL BR_INST_RETIRED.NOT_TAKEN BR_INST_EXEC.TAKEN_INDIRECT_JUMP_NON_CALL_RET BR_INST_RETIRED.NEAR_CALL_R3 MEM_LOAD_UOPS_L3_HIT_RETIRED.XSNP_NONE_PS L2_RQSTS.ALL_RFO L2_LINES_OUT.DEMAND_DIRTY L2_TRANS.L2_WB L2_RQSTS.DEMAND_DATA_RD_HIT L2_RQSTS.DEMAND_DATA_RD_MISS L1D_PEND_MISS.PENDING_CYCLES L2_RQSTS.ALL_PF
SR4: File Hiding	SSDT Hooking	ICACHE.IFETCH_STALL BR_INST_EXEC.TAKEN_INDIRECT_JUMP_NON_CALL_RET MEM_LOAD_UOPS_L3_HIT_RETIRED.XSNP_NONE_PS L2_RQSTS.ALL_RFO L2_LINES_OUT.DEMAND_DIRTY L2_TRANS.L2_WB L2_RQSTS.DEMAND_DATA_RD_HIT L2_RQSTS.DEMAND_DATA_RD_MISS L1D_PEND_MISS.PENDING_CYCLES
SR5: Direct Kernel Object Manipulation	DKOM	None found

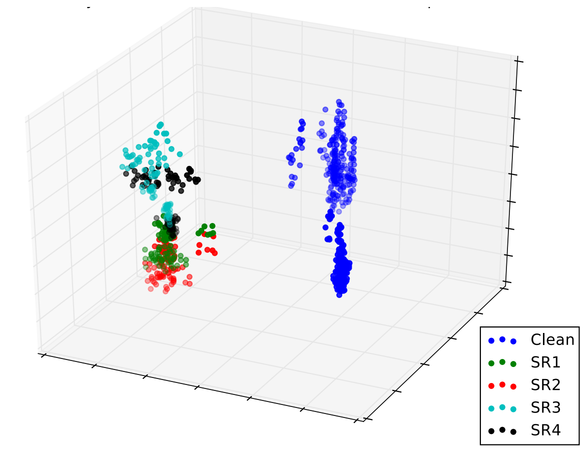


Figure 4: Visualization of the synthetic rootkit traces in a PCA reduced feature space

der to reduce the feature space from 16 down even further to three dimensions, albeit at a loss of accuracy. This allows us to graph an estimate of the traces. Fig. 4 shows these results for SR1-SR4 as well as clean traces. As can be seen, there is a very clear delineation between the clean traces and those from the synthetic rootkits. This leads us to believe that the detection of hooking rootkits using HPCs will be very accurate.

4. ROOTKIT DETECTION USING HPCS

Now that we have identified the top HPCs for detecting various types of rootkit functionality in our synthetic tests, can we use that information to detect real rootkits?

In order to answer this question, we trained a machine learning based detector using a set of clean traces as well as the dirty traces taken from the synthetic rootkits, but only for the 16 most significant HPCs determined in Section 3.5. We then used this trained detector to classify dirty traces taken from real rootkits as well as additional clean traces captured under the four different background workload conditions of Table 2.

4.1 Rootkit Samples

As examples of real rootkits, we identified 20 variants of five different well-known Windows 7 rootkits: Zeus, ZeroAccess, Hickit, Ramnit and Turla. All 100 samples were downloaded from VirusTotal [28]. We chose these rootkits because they were accessible, successfully executed on Windows 7, performed some sort of rootkit activity, and enough variants were available to collect a variety of traces. The types of rootkit attacks performed vary between the various rootkits. ZeroAccess, for example, employs IRP hooking in order to support file hiding. On the other hand, Zeus and some of its variants displayed signs of SSDT Hooking. The execution and proper functioning of all rootkit samples was manually verified.

4.2 Additional Trace Collection

Given the new rootkits, traces were collected of each rootkit sample using the 16 most significant HPCs discovered in Section 3.5. In order to further vary the testing conditions, an additional two background conditions (Table 4) were added. For each of the 100 rootkit variants, we collected 10 traces under each of the four background workload conditions for a total of 4000 dirty traces. Given that we collected far fewer HPCs, this data collection only required 8 days of CPU time in order to get all 4000 traces.

We also collected an additional 24,000 clean traces using the same 16 HPCs under the four background workload conditions.

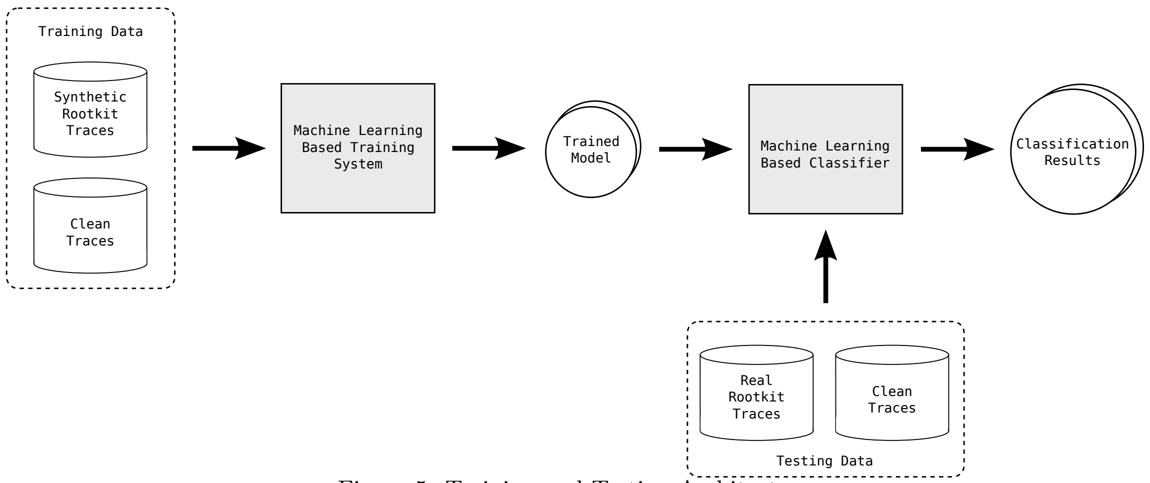


Figure 5: Training and Testing Architecture

Table 4: Additional Trace Background Workload Conditions

Name	Description
Noisy 2	The profiling benchmark is executed and Google Chrome is running, opening a variety of websites in multiple tabs.
Noisy 3	The profiling benchmark is executed and the Windows System Assessment Tool is executed to benchmark the memory and disk performance of the machine.

4.3 Machine Learning Methodology

We made use of the scikit-learn [20] Python library to implement our system.

We evaluated the effectiveness of four different classifiers for detecting rootkits. When using machine learning for classification, classifiers are used to distinguish data points in order to determine which of the N classes every point belongs to. In our case, we have used two classes - clean and dirty (infected). Hence, we can use our classifiers to predict the probability of each data point in our test data which represents the likelihood of it being a rootkit. What follows is a brief description of each of the classifiers used.

SVM Support vector machines, SVM, are based on supervised machine learning models. The classifier is trained by passing it a set of data points with each data point marked into one of the two categories. The training algorithm then finds an appropriate plane/hyperplane that separates the two classes best. To classify, the new point is mapped to either class based on its location with respect to the plane/hyperplane. We used two different SVM kernels: Linear and RBF. Linear attempts to find a simple, linear plane/hyperplane separating the classes, while RBF builds a more a complex plane/hyperplane.

OC-SVM One class SVM is an unsupervised learning model, meaning it is trained only one class of data (in this case, dirty). Unlike SVM, OC-SVM classifies data points as either belonging to the class or not. Instead of finding the best plane/hyperplane to separate data, the data points are enclosed in a distinguishing shape that contains all the data points that belong to one class. Data points outside that shape are marked as not part of the class.

Naive Bayes Naive Bayes is a supervised algorithm based on a probabilistic classification approach. In this classifier, Bayes' theorem is used to construct a probability model while assuming that all features are independent. A simple decision rule is then applied to the probability model, creating the classifier.

Decision Trees A decision tree is used in a supervised classification environment. A decision tree is made up of several features. To train the classifying tree, the training data is divided into subsets based on a given feature. Each subset is then recursively divided and checked if it adds any value to the classification process. If either the maximum depth is reached or the division no longer makes the prediction better, the node terminates. In order to classify a new data point, the point traverses the whole tree based on the branch conditions. It stops once it reaches a terminating node which specifies its predicted class.

4.4 Training and Testing

We trained our classifiers using the 500 dirty traces from the synthetic rootkits captured in Section 3 (reduced to only the 16 HPCs) as well as 20,000 of the 24,000 clean traces captured in Section 4.2. The architecture of our training approach can be seen in Fig. 5.

For machine learning techniques that require tuned parameters (such as the γ value for SVM with an RBF kernel), we applied a cross-validation grid-search over a range of possible values. The parameter that is selected is the one that gives the maximum accuracy on the test part of the cross-validated data. The parameter values are obtained by cross-validating the training data using a 60%/40% split. None of the testing data is used as part of tuning.

Because our training data contains significantly more clean traces than dirty ones, we have adjusted the weights given to each class to balance the frequencies. In this method, class samples get weights that are inversely proportional to the class size. For example, in our training data the clean class is 40 times larger than the dirty class. This can lead to skewed results if this is not corrected for. By adjusting weights of both classes during training, we can account for this discrepancy and produce more accurate results.

It is important to emphasize that our dirty training data consists of only traces from the synthetic rootkits of Section 3. Traces from the real rootkits were not included.

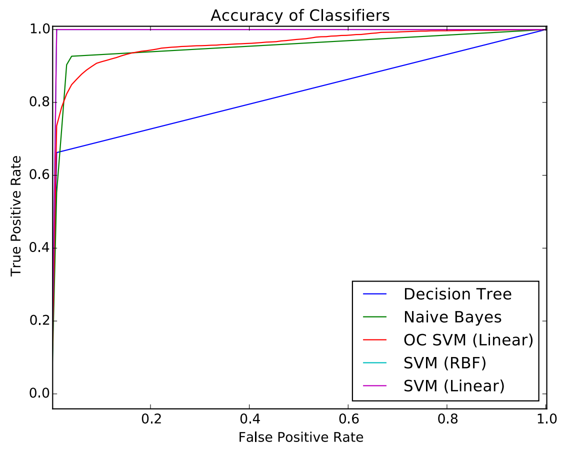


Figure 6: ROC Curve Graph

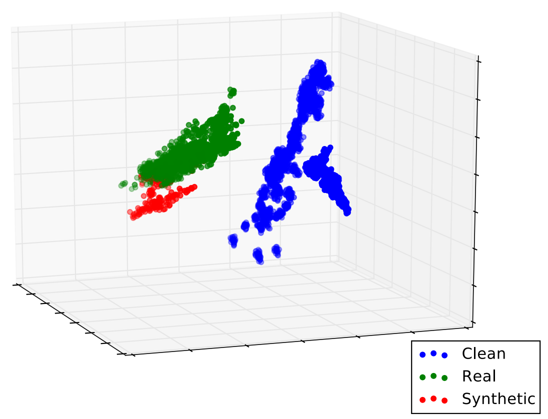


Figure 7: Visualization of the real and synthetic rootkit traces in a PCA reduced feature space

This is because we want to determine if HPCs can be used to detect previously unseen rootkits based on their functionality.

When testing our model, we used the remaining 4,000 clean traces (from the original 24,000 captured) as well as the 4,000 traces captured from the real rootkits.

4.5 Results

Table 5 shows the true positive, false positive, true negative, and false negative rates that each classifier achieves when classifying the 8000 traces found in the testing set. Fig. 6 shows the ROC curves for the classifiers as well. For classifiers that involve randomness in their execution, the data reflects the average of 50 runs.

From the results, we can see that both versions of SVM (Linear and RBF) produce extremely accurate results with a true positive rate of 99.91% and a 0% false positive rate. This indicates that the distinction between clean and dirty in the data is very clear, and hence SVM is able to easily distinguish between the two.

In order to visualize this distinction, we combined the synthetic rootkit traces, real rootkit traces, and a set of clean traces and once again performed a PCA reduction of the feature space into three dimensions. The results can be found in Fig. 7. As can be seen, there is a clear delineation between the various rootkit traces and the clean traces. This leads to very accurate detection using SVM.

The extremely high detection rate is particularly surprising given that the detector was trained on the synthetic rootkits and used to detect the real rootkits. The system is able to detect rootkits it has never seen before, even as variants. This means that HPCs are suitable for the detection of zero-day rootkit attacks as long as those rootkits employ previously known attack mechanisms.

5. DISCUSSION

The results in this paper raise a number of points that deserve further discussion.

5.1 Explanation of Significant HPCs

In some ways the results of this work leave the reader wanting because while it describes the observation and application of a phenomenon (namely that HPCs can be used to accurately detect rootkits) it does not address the question of *Why?*

When analyzing the 16 HPCs identified in Table 3, it is tempting to try and explain exactly why each HPC is significant to each rootkit. For example, one could theorize that SR1 and SR2 impact the `BR_INST_RETIRED.NEAR_TAKEN` HPC because IRP hooking causes additional branches to occur, and hence there are additional branches retired. (That explanation, while sounding good, is completely fabricated.) Indeed, the authors wrote multiple attempts at making just these sorts of explanations. However, after multiple revisions and detailed analysis of the data we removed such explanations and have instead come to a different conclusion: We do not have enough data to properly ascertain why a particular HPC is impacted by the rootkit.

To do so would require a much finer granularity of HPC collection than was employed here. One way to obtain data at that granularity would be to instrument the OS kernel to capture HPC information at various points (or perhaps at every instruction) along the control-flow, and use the data to create an annotated version of the code that details how HPCs are impacted during execution. This is effectively superimposing the time series HPC information onto a control-flow graph. (The need to directly correlated HPC changes with the code that caused them is the reason that a simple time series HPC capture could not be used.) These annotated control-flow traces could then be obtained for infected and non-infected runs, and the results compared to determine exactly which code causes the HPC data to deviate. This would allow one to much more conclusively describe why the various HPCs are impacted by the rootkits, and would also give additional insight into the possibility of rootkits designed to evade such detection techniques.

5.2 Design Considerations for a Practical Detector

Obviously the approach used to gather HPCs in this work would not be suitable for constructing a true rootkit detector. First, the trusted computing based (TCB) is too large and actually includes the very operating system kernel that malware is infecting. Intel's VTune, while convenient for gathering traces, runs at a privilege level that would allow a rootkit attacker to disable it or modify its results at runtime. In addition, our results indicate that rootkit detection

Table 5: Accuracy Results of Various Machine Learning Algorithms

ML algorithm	**TP Rate**	**FP Rate**	**TN Rate**	**FN Rate**
Decision Tree	67.43%	0.001%	99.99%	32.57%
Naive Bayes	51.22%	0.52%	99.48%	48.78%
OC-SVM	100%	49.81%	50.19%	0%
SVM(RBF Kernel)	99.88%	0%	100%	0.12%
SVM(Linear Kernel)	99.91%	0%	100%	0.09%

is most effective with at least 16 different HPCs, while modern Intel processors only allow four HPCs to be collected simultaneously.

We will now discuss the considerations that should be made for an actual rootkit detection system based on HPCs. Existing work [6, 27, 18] has already proposed a variety of design choices and recommendations, and ours will build on these. In general, an HPC based rootkit detector should not be constructed independently, instead it should be part of the integrated design of a more general purpose HPC based malware detector.

Simultaneous HPC Capture Both [6] and [27] propose that hardware should be modified to allow for more than four HPCs to be monitored simultaneously, but neither work provides guidelines for how many this should be. Our results indicate that 16 might be the lower bound, although in practice we do not believe that many more than this should be required. Existing work has obtained very good results with only the four, and with better machine learning approaches we believe that our count of 16 could be reduced as well.

Location and Updating of Detection Engine In [18] the detection engine is designed and implemented in hardware, and while both [6] and [27] place it in software, [6] proposes updates to it occur with hardware validation assistance. In general, we feel that the less reliant on hardware the detection engine is, the more robust it will be as attacks evolve and the machine learning techniques required evolve with them. Recent advances in compartmental execution provide an elegant solution to this problem. Solutions such as Intel's SGX [1] and Iso-X [7] provide methods for running code in a hardware protected enclave whose data cannot be manipulated by other software layers, including more privileged ones. This would allow a software-only detection engine that can be updated without special purpose hardware assistance beyond what is offered by the compartment system.

Secure Acquisition of HPC Data All three existing recommendations propose tamper-proof, interrupt-less capture of HPC data. We expand this recommendation by noting that the secure delivery of HPC data should be incorporated into the design of the compartment system, eliminating the need for dedicating an isolated core [6] or isolated bus going to that core.

Profiling Benchmark The profiling benchmark used in this work could, for the most part, be used as part of a practical detector. The benchmark would be executed periodically and the captured HPC values run through the rootkit detector. While a rootkit could detect that the benchmark is running and disable its functionality prior to execution, this would have a side-effect of ensuring that the benchmark is able to collect the actual, unmodified state (such as processes and network connections) of the system. Overall, If the rootkit detects that the benchmark is running and continues to hide its presence, then it can be detected using the HPC values. If the rootkit instead disables its hiding techniques in order to evade HPC detection, then the data collected by the benchmark will reveal the very things the rootkit is trying to hide, hence allowing traditional detection techniques to be used.

5.3 Evasion Techniques

The experiments performed in this work were done under the assumption that the rootkit is not aware that it will be profiled in this way. However, can rootkits adapt to evade this technique using a mimicry attack [27]? The answer to this depends on exactly how the rootkits impact the HPCs. If the HPC impacts are due to things the rootkit cannot change, such as the branches related to hooking, then the answer is likely no. However, if the HPC impacts are simply a reflection of the fact that different code with different HPC characteristics is running when infected than when not infected, then the answer might be yes. A rootkit author could modify their code to maintain a similar average of HPC effects to that of the normal OS code. However, this sort of attack would necessarily increase the number of instructions executed (potentially significantly), which makes it prone to detection by a simpler approach, such as the one employed by NumChecker [29]. A deeper discussion of mimicry attacks is available in [27].

While our experiments reveal that HPCs are effective for detecting rootkits that make use of hooking, the revelation that the DKOM based kit did not produce any significant HPCs is a sign that HPCs are not a panacea for rootkit detection. We hypothesize that there may be other types of rootkits, such as those making use of return-oriented programming [10], which may also not be detected by this technique. However, it is heartening to observe both that the vast majority of rootkits do employ hooking, and that it is still not clear if non-hooking based attacks can be as powerful [22] as their hooking counterparts. It seems unlikely we will see them equal the functionality required by modern attacks.

5.4 HPC Collection Methodology

In this work we collected HPCs inside a virtual machine using Intel's VTune, a tool primarily designed to assist developers in optimizing their programs. Other works developed a custom HPC collection mechanism [27] and/or ran directly on bare hardware [6]. While the detection results in those and this work indicate that the HPC collection was effective, the question remains regarding how much noise is introduced by the various techniques. It would be interesting to benchmark various HPC collection techniques in order to gauge their accuracy. It will also be important going forward to verify that HPC collection from within a VM has similar levels of accuracy when compared to captures done on bare metal.

6. RELATED WORK

While the original rootkit results from Demme et al. [6] (previously discussed in Section 2.3) were not very promising, our work shows significantly higher accuracy when detecting rootkits. We believe this difference in results can be best explained by looking at the rootkits tested in each work. Their experimentation included only two different real-world rootkits (one user-level and one kernel-level), while ours includes 100 variants of 5 real-world kernel rootkits. In addition, our 5 synthetic rootkits cover a variety of kernel rootkit attack mechanisms, while the kernel rootkit they tested with only employs one mechanism. In short, our testing included a more comprehensive sample of kernel rootkit techniques. This is not meant to be critical of their approach, instead this work simply provides a much more thorough focus on rootkits while their work focused on a broad range of malware. The fact that our testing was performed on Windows while theirs was performed on Linux may also have some impact, and future work should investigate those differences.

Very similar in principal to our work is Numchecker [29], a system that detects Linux rootkits by looking for HPC deviations during the execution of kernel functions. Their approach is to count the number of retired instructions, retired returns, and retired branches that occur during system calls and compare those numbers to known good values for that OS. The main difference between their work and ours can be summarized as a manual vs machine learning based approach. In Numchecker, the HPCs used for analysis were manually chosen by the authors, while in our work we apply machine learning techniques to determine the most significant HPCs. In addition, Numchecker detects rootkits by evaluating whether the HPCs values collected during the execution of a given kernel function deviate from experimentally chosen values. Our work, in contrast, applies machine learning to determine whether or not a rootkit is present. In general, our work is a more systematic study of the impact of rootkits on HPCs, and provides a more general approach for using HPCs for rootkit detection.

A number of other works have also focused on using HPCs for the detection of malware. Tang et al. [27] use unsupervised machine learning to build profiles of normal application's HPC patterns, and then detect deviations. Their focus was on detecting user-level malware during exploitation (as opposed to after infection). They demonstrated their technique by detecting attacks against real vulnerabilities in Internet Explorer 9, Adobe Reader, and Adobe Flash. They achieved over 99% accuracy on detecting the exploitation of these applications. Ozsoy et al. [18] propose the Malware Aware Processor (MAP), a hardware approach which uses the same type of micro-architecture events measured by HPCs in order to detect user-level malware in hardware.

The use of other microarchitectural features to detect malicious activity has been studied as well. kBouncer [19] uses the last branch recording (LBR) of Intel microprocessors to detect the execution of ROP [24] code. A variety of works [4, 23, 32] have investigated the use of opcodes for the detection of malware.

HPCs have also been applied in other ways to security. Maurice et al. [15] use HPCs to reverse engineer the last level cache in modern Intel processors, simplifying side-channel attacks and covert channels. Malone et al. [14] design a method for using HPCs to provide integrity checking of running applications.

7. CONCLUSION

In this work we provide an analysis of the applicability of hardware performance counters to the detection of kernel rootkits. We effectively extend and expand on the preliminary results found in Demme et al. [6], demonstrating that HPCs can be used to detect rootkits with very high accuracy (>99%). We use machine learning to identify the 16 most significant HPCs for detecting Windows 7 rootkits that make use of IRP and SSDT hooking to perform their attacks. We also demonstrate that an SVM based classifier can be trained to detect new, real-world rootkits despite only being trained on a set of synthetic rootkits with limited, but specific, functionality. This work provides many of the theoretical and practical underpinnings that will be required in order to build a fully functional, HPC-based rootkit detector.

Acknowledgments

This paper was made possible by NPRP grants 4-1593-1-260 and 8-1474-2-626 from the Qatar National Research Fund (a member of Qatar Foundation). The statements made herein are solely the responsibility of the authors.

The authors would also like to thank Aisha Hasan as well as the reviewers for their helpful comments on this work.

8. REFERENCES

[1] Intel$^{®}$ Software Guard Extensions Programming Reference, 2014. Accessed Apr. 2016 at https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf.

[2] S. Bandyopadhyay. A Study on Performance Monitoring Counters in x86-Architecture. *Indian Statistical Institute*, 2004.

[3] R. Berrendorf and H. Ziegler. PCL–the Performance Counter Library: A Common Interface to Access Hardware Performance Counters on Microprocessors, Version 1.3, 1998.

[4] D. Bilar. Opcodes as Predictor for Malware. *International Journal of Electronic Security and Digital Forensics*, 1(2):156–168, 2007.

[5] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A Portable Programming Interface for Performance Evaluation on Modern Processors. *International Journal of High Performance Computing Applications*, 14(3):189–204, 2000.

[6] J. Demme, M. Maycock, J. Schmitz, A. Tang, A. Waksman, S. Sethumadhavan, and S. Stolfo. On the Feasibility of Online Malware Detection with Performance Counters. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA 2013)*, 2013.

[7] D. Evtyushkin, J. Elwell, M. Ozsoy, D. Ponomarev, N. A. Ghazaleh, and R. Riley. Iso-x: A flexible architecture for hardware-managed isolated execution. In *47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 190–202. IEEE, 2014.

[8] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The WEKA Data Mining Software: An Update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.

[9] G. Hoglund and J. Butler. *Rootkits: Subverting the Windows kernel*. Addison-Wesley Professional, 2006.

[10] R. Hund, T. Holz, and F. C. Freiling. Return-Oriented Rootkits: Bypassing Kernel Code Integrity Protection Mechanisms. In *USENIX Security Symposium*, pages 383–398, 2009.

[11] Intel Corporation. Intel®VTune Amplifier 2015. https://software.intel.com/en-us/intel-vtune-amplifier-xe. Last accessed January 2016.

[12] K.-J. Lee and K. Skadron. Using Performance Counters for Runtime Temperature Sensing in High-Performance Processors. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, pages 8–pp. IEEE, 2005.

[13] K. London, S. Moore, P. Mucci, K. Seymour, and R. Luczak. The PAPI Cross-Platform Interface to Hardware Performance Counters. In *Department of Defense Users' Group Conference Proceedings*, pages 18–21, 2001.

[14] C. Malone, M. Zahran, and R. 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.

[15] C. Maurice, N. Scouarnec, C. Neumann, O. Heen, and A. Francillon. Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters. In *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID 2015)*, pages 48–65. Springer International Publishing, 2015.

[16] J. M. May. MPX: Software for Multiplexing Hardware Performance Counters in Multithreaded Programs. In *Proceedings of the 15th International Parallel and Distributed Processing Symposium*. IEEE, 2001.

[17] Microsoft Corporation. Introduction to File System Filter Drivers. https://msdn.microsoft.com/en-us/windows/hardware/drivers/ifs/introduction-to-file-system-filter-drivers. Last Accessed February 2017.

[18] M. Ozsoy, C. Donovick, I. Gorelik, N. Abu-Ghazaleh, and D. Ponomarev. Malware-Aware Processors: A Framework for Efficient Online Malware Detection. In *IEEE 21st International Symposium on High Performance Computer Architecture (HPCA 2015)*, pages 651–661, 2015.

[19] V. Pappas, M. Polychronakis, and A. D. Keromytis. Transparent ROP Exploit Mitigation Using Indirect Branch Tracing. In *USENIX Security*, pages 447–462, 2013.

[20] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[21] J. Rhee, R. Riley, D. Xu, and X. Jiang. Defeating Dynamic Data Kernel Rootkit Attacks via VMM-based Guest-Transparent Monitoring. In *Proceedings of International Conference on Availability, Reliability and Security (ARES)*, pages 74–81. IEEE, 2009.

[22] R. Riley. A Framework for Prototyping and Testing Data-Only Rootkit Attacks. *Computers and Security*, 37(0):62 – 71, 2013.

[23] I. Santos, F. Brezo, J. Nieves, Y. K. Penya, B. Sanz, C. Laorden, and P. G. Bringas. Idea: Opcode-sequence-based Malware Detection. In *Engineering Secure Software and Systems*, pages 35–43. Springer, 2010.

[24] H. Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, pages 552–561. ACM, 2007.

[25] K. Singh, M. Bhadauria, and S. A. McKee. Real Time Power Estimation and Thread Scheduling via Performance Counters. *ACM SIGARCH Computer Architecture News*, 37(2):46–55, 2009.

[26] B. Sprunt. The Basics of Performance-Monitoring Hardware. *IEEE Micro*, pages 64–71, 2002.

[27] A. Tang, S. Sethumadhavan, and S. J. Stolfo. Unsupervised Anomaly-Based Malware Detection Using Hardware Features. In *Proceedings of Research in Attacks, Intrusions and Defenses (RAID 2014)*, 2014.

[28] VirusTotal. VirusTotal-Free Online Virus, Malware and URL Scanner. https://www.virustotal.com/. Last accessed February 2016.

[29] X. 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. IEEE, 2013.

[30] Z. Wang, X. Jiang, W. Cui, and P. Ning. Countering Kernel Rootkits with Lightweight Hook Protection. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 545–554, New York, NY, USA, 2009. ACM.

[31] V. M. Weaver and S. McKee. Can Hardware Performance Counters be Trusted? In *IEEE International Symposium on Workload Characterization (IISWC 2008)*, pages 141–150. IEEE, 2008.

[32] G. Yan, N. Brown, and D. Kong. Exploring Discriminatory Features for Automated Malware Classification. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 41–61. Springer, 2013.