Unsupervised Anomaly-Based Malware Detection Using Hardware Features

Adrian Tang, Simha Sethumadhavan, and Salvatore J. Stolfo

Columbia University, New York, USA
{atang,simha,sal}@cs.columbia.edu

Abstract. Recent works have shown promise in detecting malware programs based on their dynamic microarchitectural execution patterns. Compared to higher-level features like OS and application observables, these microarchitectural features are efficient to audit and harder for adversaries to control directly in evasion attacks. These data can be collected at low overheads using widely available hardware performance counters (HPC) in modern processors. In this work, we advance the use of hardware supported lower-level features to detecting malware exploitation in an anomaly-based detector. This allows us to detect a wider range of malware, even zero days. As we show empirically, the microarchitectural characteristics of benign programs are noisy, and the deviations exhibited by malware exploits are minute. We demonstrate that with careful selection and extraction of the features combined with unsupervised machine learning, we can build baseline models of benign program execution and use these profiles to detect deviations that occur as a result of malware exploitation. We show that detection of real-world exploitation of popular programs such as IE and Adobe PDF Reader on a Windows/x86 platform works well in practice. We also examine the limits and challenges in implementing this approach in face of a sophisticated adversary attempting to evade anomaly-based detection. The proposed detector is complementary to previously proposed signature-based detectors and can be used together to improve security.

Keywords: Hardware Performance Counter, Malware Detection.

1 Introduction

Malware infections have plagued organizations and users for years, and are growing stealthier and increasing in number by the day. In response to this trend, defenders have created commercial antivirus (AV) protections, and are actively researching better ways to detect malware. An emerging and promising approach to detect malware is to build detectors in hardware [3]. The idea is to use information easily available in hardware (typically via HPC) to detect malware. It has been argued that hardware malware schemes are desirable for two reasons: first, unlike software malware solutions that aim to protect vulnerable software with equally vulnerable software[1], hardware systems protect vulnerable software with

[1] Software AV systems roughly have the same bug defect density as regular software.

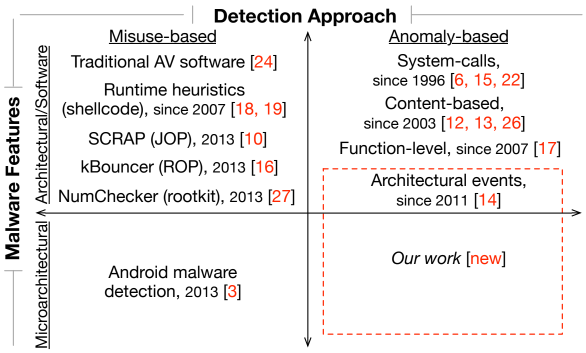
Detection Approach
Misuse-based
Anomaly-based
Malware Features
Architectural/Software
Microarchitectural
Traditional AV software [24]
Runtime heuristics (shellcode), since 2007 [18, 19]
SCRAP (JOP), 2013 [10]
kBouncer (ROP), 2013 [16]
NumChecker (rootkit), 2013 [27]
Android malware detection, 2013 [3]
System-calls, since 1996 [6, 15, 22]
Content-based, since 2003 [12, 13, 26]
Function-level, since 2007 [17]
Architectural events, since 2011 [14]
Our work [new]

Fig. 1. Taxonomy of malware detection approaches and some example works

robust hardware implementations that have lower bug defect density because of their simplicity. Second, while a motivated adversary can evade either defense, evasion is harder in a system that utilizes hardware features. The intuition is that the attacker does not have the same degree of control over lower-level hardware features as she has with software ones. For instance, it is easier to change system calls or file names than induce cache misses or branch misprediction in a precise way across a range of time scales while exploiting the system.

In this paper we introduce techniques to advance the use of lower-level microarchitectural features in the anomaly-based detection of malware exploits. Existing malware detection techniques can be classified along two dimensions: *detection approach* and the *malware features* they target, as presented in Figure 1. Detection approaches are traditionally categorized into misuse-based and anomaly-based detection. Misuse-based detection flags malware using pre-identified attack signatures or heuristics. It can be highly accurate against known attacks but can be easily evaded with slight modifications that deviate from the signatures. On the other hand, anomaly-based detection characterizes baseline models of normalcy state and identifies attacks based on deviations from these models. Besides known attacks, it can potentially identify novel ones. There are a range of features that can be used for detection: until 2013, they were OS and application-level observables such as system calls and network traffic. Since then, lower-level features closer to hardware such as microarchitectural events have been used for malware detection. Shown in Figure 1, we examine for the first time, the feasibility and limits of anomaly-based malware detection using both architectural and low-level microarchitectural features available from HPCs.

Prior misuse-based research that uses microarchitectural features such as [3] focuses on flagging Android malicious apps by detecting payloads. A key distinction between our work and prior work is *when* the malware is detected. Malware infection typically comprises two stages, exploitation and take-over. In the exploitation stage, an adversary exercises a bug in the victim program to hijack control of the program execution. Exploitation is then followed by more elaborate take-over procedures to run a malicious payload such as a keylogger.

Our work focuses on detecting malware during exploitation, as it not only gives more lead time for mitigations but can also act as an early-threat detector to improve the accuracy of subsequent signature-based detection of payloads.

The key intuition for the anomaly-based detection of malware exploits stems from the observation that the malware, during exploitation, alters the original program flow to execute peculiar non-native code in the context of the victim program. Such unusual code execution tend to cause perturbations to the dynamic execution characteristics of the program. If these perturbations are observable, they can form the basis of detecting malware exploits.

In this work, we model the baseline characteristics of common vulnerable programs – Internet Explorer 8 and Adobe PDF Reader 9 (two of the most attacked programs) and examine if such perturbations do exist. Intuitively one might expect the deviations caused by exploits to be fairly small and unreliable, especially in vulnerable programs with extremely varied use such as in the ones we study. This intuition is validated in our measurements. On a Windows system using Intel x86 chips, our experiments indicate that distributions of measurements from the hardware performance counters are positively skewed, with many values being clustered near zero. This implies minute deviations caused by the exploit code cannot be effectively discerned directly. However, we show that this problem of identifying deviations from the heavily skewed distributions can be alleviated. We show that by using power transform to amplify small differences, together with temporal aggregation of multiple samples, we can identify the execution of the exploit within the context of the larger program execution. Further, in a series of experiments, we systematically evaluate the detection efficacy of the models over a range of operational factors, events selected for modeling and sampling granularity. For IE exploits, we can identify 100% of the exploitation epochs with 1.1% false positives. Since exploitation typically occurs across nearly 20 epochs, even with a slightly lower true positive rate, we can detect exploits with high probability. These are achieved at a sampling overhead of 1.5% slowdown using sampling rate of 512K instructions epochs.

Further we examine the resilience of our detection technique to evasion strategies of a more sophisticated adversary. We model *mimicry* attacks that craft malware to exhibit event characteristics that resemble normal code execution to evade our anomaly detection models. With generously optimistic assumptions about attacker and system capabilities, we demonstrate that the models are susceptible to the mimicry attack. In a worst case scenario, the detection performance deteriorates by up to 6.5%. Due to this limitation we observe that anomaly detectors cannot be the only defensive solution but can be valuable as part of an ensemble of detectors that can include signature-based ones.

The rest of the paper is organized as follows. We provide a background on modern malware exploits in Section 2. We detail our experimental setup in Section 3. We present our approach in building models for the study in Section 4, and describe the experimental results in Section 5. Section 6 examines evasion strategies of an adaptive adversary and their impact on detection performance. Section 7 discusses related work, and we conclude in Section 8.

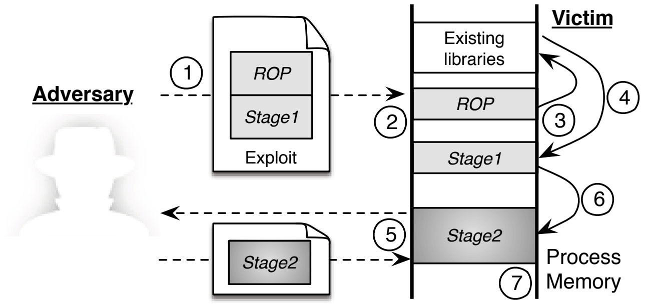

Fig. 2. Multi-stage exploit process

2 Background

Figure 2 shows a typical multi-stage malware infection process that results in a system compromise. The necessity for its multi-stage nature will become clear as we explain the exploit process in this section.

Triggering the Vulnerability. First the adversary crafts and delivers the exploit to the victim to target a specific vulnerability known to the adversary (Step ①). The vulnerability is in general a memory corruption bug; the exploit is typically sent to a victim from a webpage or a document attachment from an email. When the victim accesses the exploit, two exploit sub-programs, commonly known as the *ROP* and *Stage1* "shellcodes", load into the memory of the vulnerable program (Step ②). The exploit then uses the vulnerability to transfer control to the *ROP* shellcode (Step ③).

Code Reuse Shellcode (*ROP*). To prevent untrusted data being executed as code, modern processors provide Data Execution Prevention (DEP) to restrict code from being run from data pages. To support JIT compilation however, DEP can be toggled by the program itself. So the *ROP*-stage shellcode typically circumvents DEP by reusing instructions in the original program binary – hence the name Code Reuse Shellcode – to craft a call to the function that disables DEP for the data page containing the next *Stage1* shellcode. The ROP shellCode then redirects execution to the next stage. (Step ④) [16].

***Stage1* Shellcode.** This shellcode is typically a relatively small – from a few bytes to about 300 bytes[2] – code stub with exactly one purpose: to download a larger (evil) payload which can be run more freely. To maintain stealth, it downloads the payload in memory (Step ⑤).

***Stage2* Payload.** The payload is the final piece of code that the adversary wants to execute on the target to perform a specific malicious task. The range of functionality of this payload, commonly a backdoor, keylogger, or reconnaissance program, is unlimited. After the payload is downloaded, the *Stage1* shellcode runs this payload as an executable using reflective DLL injection (Step ⑥), a

[2] As observed at `http://exploit-db.com`

stealthy library injection technique that does not require any physical files [5]. By this time, the victim system is fully compromised (Step ⑦).

The *Stage1* shellcode and *Stage2* payload are different in size, design and function, primarily due to the operational constraints on the *Stage1* shellcode. When delivering the initial shellcode in the exploit, exploit writers typically try to use as little memory as possible to ensure that the program does not unintentionally overwrite their exploit code in memory. To have a good probability for success, this code needs to be small, fast and portable, and thus is written in assembly language and uses very restrictive position-independent memory addressing style. These constraints limit the adversary's ability to write very large shellcodes. In contrast, the *Stage2* payload does not have all these constraints and can be developed like any regular program. This is similar to how OSes use small assembly routines to bootstrap and then switch to compiled code.

The strategy and structure described above is representative of a large number of malware especially those created with recent web exploit kits [25]. These malware exploits execute completely from memory and in the process context of the host victim program. Further, they maintain disk and process stealth by ensuring no files are written to disk and no new processes are created, and thus easily evade most file based malware detection techniques.

3 Experimental Setup

Do the execution of different shellcode stages exhibit observable deviations from the baseline performance characteristics of the user programs? Can we use these deviations, if any, to detect a malware exploit as early as possible in the infection process? To address these questions, we conduct several feasibility experiments, by building baseline per-program models using machine learning classifiers and examining their detection efficacy over a range of operational factors. Here, we describe our experimental setup and detail how we collect and label the measurements attributed to different malware exploit stages.

3.1 Exploits

Unlike SPEC, there are no standard exploit benchmarks. We rely on a widely-used penetration testing tool *Metasploit* (from `www.metasploit.com`) to generate exploits for common vulnerable programs from publicly available information. We use exploits that target the security vulnerabilities *CVE-2012-4792*, *CVE-2012-1535* and *CVE-2010-2883* on IE 8 and the web plug-ins, *i.e.,* Adobe Flash 11.3.300.257 and Adobe Reader 9.3.4 respectively. We choose to utilize *Metasploit* because the exploitation techniques it employs in the exploits are representative of multi-stage nature of real-world exploits.

Besides targeting different vulnerabilities using different ROP shellcode from relevant library files (`msvcrt.dll`, `icucnv36.dll`, `flash32.ocx`), we also vary both the *Stage1* (reverse_tcp, reverse_http, bind_tcp) shellcode and the *Stage2* final payload (meterpreter, vncinject, command_shell) used in the exploits.

Additionally, we instrument the start and end of the respective malware stages with debug trap *int3* instructions (`0xCC`) of one byte long, to label the exploit measurements with the respective stages solely for evaluation purposes.

3.2 Measurement Infrastructure

Since most real-world exploits run on Windows and PDF readers, and none of the architectural simulators can run programs of this scale, we use measurements from production machines. We develop a Windows driver to configure the performance monitoring unit on Intel i7 2.7GHz IvyBridge Processor to interrupt once every N instructions and collect the event counts from the HPCs. We also record the Process ID (PID) of the currently executing program so that we can filter the measurements based on processes.

We collect the measurements from a VMware Virtual Machine (VM) environment, installed with Windows XP SP3 and running a single-core with 512MB of memory. With the virtualized HPCs in the VM, this processor enables the counting of two fixed events (clock cycles, instruction retired) and up to a limit of four events simultaneously. We configure the HPCs to update the event counts only in the user mode. To ensure experiment fidelity for the initial study, measurements from the memory buffer are read and transferred via TCP network sockets to a recording program deployed in another VM. This recording program saves the stream of measurements in a local file that is used for our analysis.

We experiment with various sampling interval of N instructions. We choose to begin the investigation with a sampling rate of every 512,000 instructions since it provides a reasonable amount of measurements without incurring too much overhead (See Section 5.4 for an evaluation of the sampling overhead). Each sample consists of the event counts from one sampling time epoch, the identifying PID and the exploit stage label.

3.3 Collection of Clean and Infected Measurements

To obtain clean exploit-free measurements for IE 8, we randomly browse websites that use different popular web plugins available on IE *viz.,* Flash, Java, PDF, Silverlight, and Windows Media Player extensions. We visit the top 20 websites from Alexa and include several other websites to widen the coverage of the use of the various plug-ins. Within the browser, we introduce variability by randomizing the order in which the websites are loaded across runs and by navigating the websites by clicking links randomly and manually on the webpages. The dynamic content on the websites also perturbs the browser caches. We use a maximum of two concurrent tabs. In addition, we simulate plug-in download and installation functions.

For Adobe PDF measurements, we download 800 random PDFs from the web, reserving half of them randomly for training and the other half for testing. To gather infected measurements, we browse pages with our PDF exploits with the same IE browser that uses the PDF plug-in. We use Metasploit to generate these

PDF exploits and ensure that both the clean and unclean PDFs have the same distribution of file types, for instance, same amount of Javascript.

We stop gathering infected measurements when we see creation of a new process. Usually the target process becomes unstable due to the corrupted memory state, and the malicious code typically "migrates" itself to another new or existing process to ensure persistence after the execution of the *Stage2* payload. This is an indication that the infection is complete.

While there are factors that may affect the results of our measurements, we take additional care to mitigate the following possible biases in our data during the measurement collection:

(1) **Between-run Contamination:** After executing each exploit and collecting the measurements, we restore the VM to the state before the exploit is exercised. This ensures the measurements collected are independent across training and testing sets, and across different clean and exploit runs.

(2) **Exploitation Bias:** Loading the exploits in the program in only one way may bias the sampled measurements. To reduce this bias, we collect the measurements while loading the exploit in different ways: (a) We launch the program and load the URL link of the generated exploit page. (b) With an already running program instance, we load the exploit page. (c) We save the exploit URL in a shortcut file and launch the link shortcut with the program.

(3) **Network Condition Bias:** The VM environment is connected to the Internet. To ensure that the different network latencies do not confound the measurements, we configure the VM environment to connect to an internally-configured *Squid* (from `www.squid-cache.org`) proxy and throttle the network bandwidth from 0.5 to 5Mbps using *Squid* delay pools. We vary the bandwidth limits while collecting measurements for both the exploit code execution and clean runs.

4 Building Models

To use HPC measurements for anomaly-based detection of malware exploits, we need to build classification models to describe the baseline characteristics for each program we protect. These program characteristics are relatively rich in information and, given numerous programs, manually building the models is nearly impossible. Instead we rely on unsupervised machine learning techniques to dynamically learn possible hidden structure in these data. We then use this hidden structure – aka model – to detect deviations during exploitation.

We rely on a class of *unsupervised* one-class machine learning techniques for model building. The one-class approach is very useful because the classifier can be trained *solely* with measurements taken from a clean environment. This removes the need to gather measurements affected by exploit code, which is hard to implement and gather in practice. Specifically, we model the characteristics with the one-class Support Vector Machine (oc-SVM) classifier that uses the non-linear Radial Basis Function (RBF) kernel. In this study, the collection of the labeled measurements is purely for evaluating the effectiveness of the models in distinguishing the measurements taken in the presence of malware code execution.

Table 1. Shortlisted candidate events to be monitored

Architectural Events	
Name	**Event Description**
LOAD	Load instructions (ins.)
STORE	Store ins.
ARITH	Arithmetic ins.
BR	Branch (br.) ins.
CALL	All near call ins.
CALL_D	Direct near call ins.
CALL_ID	Indirect near call ins.
RET	Near return ins.

Microarchitectural Events	
Name	**Event Description**
LLC	Last level cache references
MIS_LLC	Last level cache misses
MISP_BR	Mispredicted br. ins.
MISP_RET	Mispred. near return ins.
MISP_CALL	Mispred. near call ins.
MISP_BR_C	Mispred. conditional br.
MIS_ICACHE	iCache misses
MIS_ITLB	iTLB misses
MIS_DTLBL	D-TLB load misses
MIS_DTLBS	D-TLB store misses
STLB_HIT	sTLB hits after iTLB misses
%MIS_LLC[a]	% of last level cache misses
%MISP_BR[a]	% of mispred. br.
%MISP_RET[a]	% of mispred. near RET ins.

[a] These *derived* events are not directly measured, but computed with two events measured by the HPCs. For example, %MISP_BR is computed as MISP_BR/BR.

4.1 Feature Selection

While the Intel processor we use for our measurements permits hundreds of events to be monitored using HPCs, not all of them are equally useful in characterizing the execution of programs. We examine most events investigated in previous program characterization works [21,9], and various other events informed by our understanding of malware behavior. Out of the hundreds of possible events that can be monitored, we shortlist 19 events for this study in Table 1. We further differentiate between the *Architectural* events that give an indication of the execution mix of instructions in any running program, and the *Microarchitectural* ones that are dependent on the specific system hardware makeup.

Events with Higher Discriminative Power. The processor is limited to monitoring up to 4 events at any given time. Even with the smaller list of shortlisted events, we have to select only a subset of events, aka features, that can most effectively differentiate clean execution from infected execution. With the collected labeled measurements, we compute the Fisher Score (*F-Score*) to provide a quantitative measure of how effective a feature can discriminate measurements in clean executions from those in infected executions. The F-Score is a widely-used feature selection metric that measures the discriminative power of features [4]. A feature with better discriminative power would have a larger separation between the means and standard deviations for samples from different classes. The F-Score measures this degree of separation. The larger the F-Score, the more discriminative power the feature is likely to have. However, a limitation to using the F-Score is that it does not account for mutual information/dependence between features, but it can guide our selection of a subset of "more useful" features. Since we are trying to differentiate samples with malicious code execution from those without, we compute the corresponding F-Scores for each event.

Table 2. Top 7 most discriminative events for different stages of exploit execution (Each event set consists of 4 event names in **Bold**. E.g, monitoring event set *A-0* consists of simultaneously monitoring Ret, Call_D, Store and Arith event counts.)

Exploit Stage	Set Label	Events ranked by F-scores						
		1	2	3	4	5	6	7
Architectural Events								
ROP	A-0	**Ret**	**Call_D**	**Store**	**Arith**	Call	Load	Call_Id
Stage1	A-1	**Store**	**Load**	**Call_ID**	**Ret**	Call_D	Call	Arith
Stage2	A-2	**Store**	**Call_ID**	**Ret**	**Call_D**	Call	Arith	Br
Microarchitectural Events								
ROP	M-0	**Misp_Br_C**	%Misp_Br	**Misp_Br**	%Misp_Ret	**Mis_Itlb**	**Mis_Llc**	Mis_Dtlbs
Stage1	M-1	**Misp_Ret**	**Misp_Br_C**	%Misp_Ret	%Misp_Br	**Mis_Dtlbs**	**Stlb_Hit**	Misp_Br
Stage2	M-2	**Misp_Ret**	**Stlb_Hit**	**Mis_Icache**	**Mis_Itlb**	%Misp_Ret	Misp_Call	Mis_Llc
Both Architectural and Microarchitectural Events								
ROP	AM-0	**Misp_Br_C**	%Misp_Br	**Misp_Br**	%Misp_Ret	**Mis_Itlb**	**Ret**	Mis_Llc
Stage1	AM-1	**Store**	**Load**	**Misp_Ret**	**Call_ID**	Ret	Call_D	Call
Stage2	AM-2	**Store**	**Call_ID**	**Misp_Ret**	**Ret**	Call_D	Call	Stlb_Hit

We compute the F-Scores for the different stages of malware code execution for each event and reduce the shortlisted events to the 7 top-ranked events for each of the two categories, as well as for the two categories combined, as shown in Table 2. Each row consists of the top-ranked events for an event category and the exploit stage.

We further select top 4 events from each row to form 9 candidate event sets that we will use to build the baseline characteristic models of the IE browser. Each model constructed with one set of events can then be evaluated for its effectiveness in the detection of various stages of malware code execution. For brevity, we assign a label (such as *A-0* and *AM-2*) to each set of 4 events in Table 2 and refer to each model based on this *set label*. We note that the derived events such as %Misp_Br are listed in the table solely for comparison. Computing them requires monitoring two events and reduces the number of features used in the models. Via experimentation, we find that using them in the models does not increase the efficacy of the models. Thus, we exclude them from the event sets.

Feature Extraction. Each sample consists of simultaneous measurements of all the four event counts in one time epoch. We convert the measurements in each sample to the vector subspace, so that each classification vector is represented as a four-feature vector. Each vector, using this feature extraction method, represents the measurements taken at the smallest time-slice for that sampling granularity. These features will be used to build *non-temporal* models.

Since we observe that malware shellcode typically runs over several time epochs, there may exist temporal relationships in the measurements that can be exploited. To model any potential temporal information, we extend the dimensionality of each sample vector by grouping the N consecutive samples and combining the measurements of each event to form a vector with $4N$ features. We use $N = 4$ to create sample vectors consisting of 16 features each, so each sample vector effectively represents measurements across 4 time epochs. By grouping samples across several time epochs, we use the synthesis of these event measurements to build *temporal* models.

With the granularity at which we sample the measurements, the execution of the ROP shellcode occurs within the span of just one sample. Since we are creating vectors with a number of samples as a group, the ROP payload will only contribute to one small portion of a vector sample. So we leave out the ROP shellcode for testing using this form of feature extraction.

5 Results

5.1 Anomalies Not Directly Detectable

We first investigate if we can gain insights into the distribution of the event counts for a clean environment and one attacked by an exploit. Without assuming any prior knowledge of the distributions, we use box-and-whisker[3] plots of normalized measurements for different events. These plots offer a visual gauge of the range and variance in the measurements and an initial indication on how distinguishable the measurements taken with the execution of different malware code stages are from the *clean* measurements from an exploit-free environment.

These distribution comparisons suggest that any event anomalies manifested by malware code execution are not trivially detectable, due to two key observations. (1) Most of the measurement distributions are very positively skewed, with many values clustered near zero. (2) Deviations, if any, from the baseline event characteristics due to the exploit code are not easily discerned.

5.2 Power Transform

To address this challenge, we rely on rank-preserving power transform on the measurements to positively scale the values. In the field of statistics, the power transform is a common data analysis tool to transform non-normally distributed data to one that can be approximated by a normal distribution. Used in our context, it has the value of magnifying any slight deviations that the malware code execution may have on the baseline characteristics.

For each event type, we find the appropriate power parameter λ such that the normalized median is roughly 0.5. For each event i, we maintain and use its associated parameter λ_i to scale all its corresponding measurements throughout the experiment. Each normalized and scaled event measurement for event i, normalized_i, is transformed from the raw value (raw_i), minimum value (min_i), maximum value (max_i) as follows:

$$\text{normalized}_i = \left(\frac{\text{raw}_i - \text{min}_i}{\text{max}_i}\right)^{\lambda_i} \tag{1}$$

[3] The box-and-whisker plot is constructed with the bottom and top of the box representing the first and third quartiles respectively. The red line in the box is the median. The whiskers extend to 1.5 times the length of the box. Any outliers beyond the whiskers are plotted as blue + ticks.

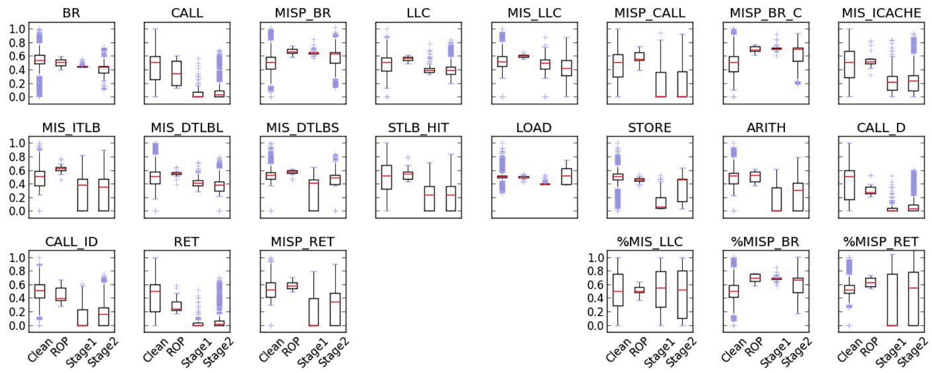

Fig. 3. Distribution of events (*after* power transform) with more discernible deviations

Using this power transform, we plot the distributions of all the events, in Figure 3. Now we observe varying deviations from baseline characteristics due to different stages of malware code execution for various event types. Some events (such as MISP_RET and STORE) show relatively larger deviations, especially for the *Stage1* exploit shellcode. These events likely possess greater discriminative power in indicating the presence of malware code execution. Clearly, there are also certain events that are visually correlated. The RET and CALL exhibit similar distributions. We can also observe strong correlation between those computed events (such as %MISP_BR) and their constituent events (such as MISP_BR).

5.3 Evaluation Metrics for Models

To visualize the classification performance of the models, we construct the *Receiver Operating Characteristic* (ROC) curves which plot the percentage of truely identified malicious samples (True positive rate) against the percentage of *clean* samples falsely classified as malicious (False positive rate). Each sample in the non-temporal model corresponds to the set of performance counter measurements in one epoch; each temporal sample spans over 4 epochs. Furthermore, to contrast the relative performance between the models in the detection of malicious samples, the area under the *ROC* curve for each model can be computed and compared. This area, commonly termed as the *Area Under Curve* (AUC) score, provides a quantitative measure of how well a model can distinguish between the clean and malicious samples for varying thresholds. The higher the AUC score, the better the detection performance of the model.

5.4 Detection Performance of Models

We first build the oc-SVM models with the training data, and evaluate them with the testing data using the non-temporal and temporal modeling on the nine event sets. To characterize and visualize the detection rates in terms of true

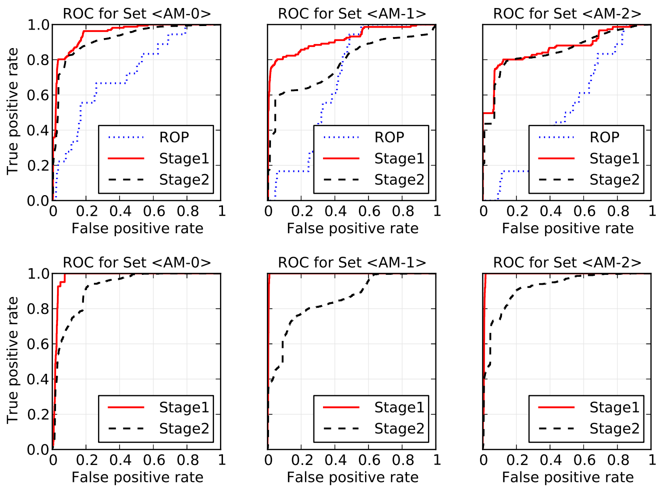

Fig. 4. Top: ROC plots for *Non-Temporal* 4-feature models for IE. **Bottom:** ROC plots for *Temporal* 16-feature models for IE.

and false positives over varying thresholds, we present the ROC curves of both approaches in Figure 4. For brevity, we only present the ROC curves for models that use both architectural and microarchitectural events. We also present the overall detection results in terms of AUC scores in Figure 5 and highlight the key observations that affect the detection accuracy of the models below.

Different Stages of Malware Exploits. We observe that the models, in general, perform best in the detection of the *Stage1* shellcode. These results suggest the *Stage1* shellcode exhibits the largest deviations from the baseline architectural and microarchitectural characteristics of benign code. We achieve a best-case detection accuracy of 99.5% for *Stage1* shellcode with *AM-1* models.

On the other hand, the models show mediocre detection capabilities for the ROP shellcode. The models does not perform well in the detection of the ROP shellcode, likely because the sampling granularity at 512k instructions is too coarse-grained to capture the deviations from the ROP shellcode in the baseline models. While the *Stage1* and *Stage2* shellcode executes within several time epochs, we measured that the ROP shellcode takes 2182 instructions on average to complete execution. It ranges from as few as 134 instructions (for the Flash ROP exploit) to 6016 instructions (for the PDF ROP exploit). Since we are keeping the sampling granularity constant, the sample that contains measurements during the ROP shellcode execution will largely consist of samples from the normal code execution.

Non-Temporal vs Temporal Modeling. We observe that the detection accuracy of the models for all event sets improves with the use of temporal information. By including more temporal information in each sample vector, we reap the benefit of magnifying any deviations that are already observable in the

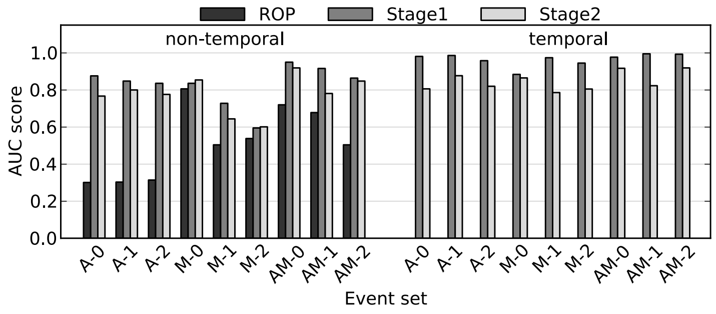

Fig. 5. Detection AUC scores for different event sets using non-temporal and temporal models for IE

non-temporal approach. For event set *M-2*, this temporal approach of building the models improves the AUC score from the non-temporal one by up to 58.8%.

Architectural vs Microarchitectural Events. We quantify the detection capabilities of our models by considering the architectural and microarchitectural features separately and in combination. Models built using only architectural events achieve AUC scores on average 4.1% better than those built solely with microarchitectural events. Combining the use of microarchitectural events with architectural ones improves the average AUC scores by 5.8% and 1.4% for microarchitectural-only and architectural-only models respectively. It is more advantageous to incorporate the use of both types of events in the detection models. For instance, by selecting and modeling both the most discriminative architectural and microarchitectural events together, we can achieve higher detection rates of up to an AUC score of 99.5% for event set *AM-1*.

Different Sampling Granularities. While we use the sampling rate of 512K instructions for the above experiments, we also examine the impact on detection efficacy for various sampling granularities. Although the hardware-based HPCs incur a near-zero overhead in the monitoring of the event counts, a pure software-only implementation of the detector still requires running programs to be interrupted periodically to sample the event counts. This inadvertently leads to a slowdown of the overall running time of programs due to this sampling overhead. To inform the deployment of a software-only implementation of such a detection paradigm, we evaluate the sampling performance overhead for different sampling rates.

To measure this overhead, we vary the sampling granularity and measure the slowdown in the programs from the SPEC 2006 benchmark suite. We also repeat the experiments using the event set *AM-1* to study the effect of sampling granularity has on the detection accuracy of the model. We plot the execution time slowdown over different sampling rates with the corresponding detection AUC scores for various malware exploit stages in Figure 6.

We observe that the detection performance generally deteriorates with coarser-grained sampling. This illustrates a key limitation of the imprecise sampling

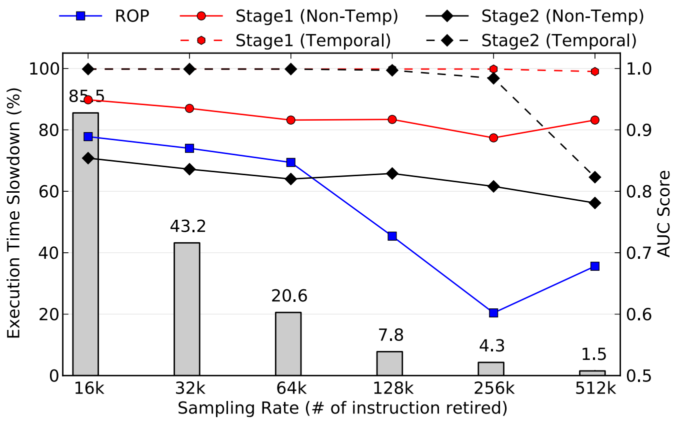

Fig. 6. Trade-off between sampling overhead for different sampling rates versus detection accuracy using set *AM-1*

technique used on Windows systems. For example, during the span of instructions retired in one sample, while we may label these measurements as belonging to a specific process PID, these measurements may also contain measurements belonging to other processes context-switched in and out during the span of this sample. The interleaved execution of different processes creates this "noise" effect that becomes more pronounced with a coarser-grained sampling rate and deteriorates the detection performance. Nonetheless, we note that the reduction in sampling overhead at coarser-grained rates far proportionately outstrips the decrease in detection performance.

Constrained Environments. To further investigate the impact of the aforementioned "noise" effect, we also assess the impact on detection accuracy in the scenario where we deploy both the online classification and the measurement gathering in the same VM. As described in Section 3.2, we collect the measurements in our study from one VM and transfer the measurements to the recorder in another VM to be saved and processed. We term this cross-remote-VM scenario where the sampling and the online classification are performed on different VMs as *R-1core*.

For this experiment, we use the event model set *AM-1* using two additional local-VM scenarios utilizing both one and two cores separately. We term these two scenarios as *L-1core* and *L-2core* respectively. We present the detection AUC scores for the three different scenarios in Table 3 (*Left*).

We observe the detection performance suffers when the online classifier is deployed locally together with the sampling driver. This may be due to possible noise introduced to the event counts while the online detector is executing and processing the stream of samples. This highlights a key limitation of the current method of periodic collection of HPC measurements on Windows systems, where we are unable to cleanly segregate the measurements on a per-process basis.

Table 3. AUC scores for: **(Left)** Constrained scenarios for IE using set *AM-1* and **(Right)** Stand-alone Adobe PDF Reader

Scenario	**Non-Temporal**			**Temporal**	
Label	**ROP**	**Stage1**	**Stage2**	**Stage1**	**Stage2**
L-1core	0.505	0.895	0.814	0.918	0.900
L-2core	0.496	0.890	0.807	0.907	0.813
R-1core	0.678	0.916	0.781	0.995	0.823

Set	**Non-Temporal**			**Temporal**	
Label	**ROP**	**Stage1**	**Stage2**	**Stage1**	**Stage2**
AM-0	0.931	0.861	0.504	0.967	0.766
AM-1	0.857	0.932	0.786	0.999	0.863
AM-2	0.907	0.939	0.756	0.998	0.912

To alleviate this problem, we envision a software-only implementation on a distributed or multi-core system in which the online detector is running separately from the system or core being protected. Furthermore, since this detection approach requires little more than a stream of HPC measurements, this makes it suitable as an out-of-VM deployment in a Virtual Machine Introspection (VMI)-based setting [30] for intrusion detection. This approach requires minimum guest data structures, relieving the need to bridge the semantic gap, a common problem faced by VMI works. Another potential avenue to alleviate the "noise" problem is a pure hardware implementation using a separate and secure dedicated core or co-processor for the execution of an online detector as proposed in [3].

5.5 Results for Adobe PDF Reader

Due to space constraints, we do not present the full results from our experiments on the stand-alone Adobe PDF Reader. We present the AUC detection performance of the models built with the event sets *AM-0,1,2* in Table 3 (*Right*). Compared to the models for IE, the detection of ROP and *Stage1* shellcode generally improves for the Adobe PDF Reader. We even achieve an AUC score of 0.999 with the *AM-1* temporal model. The improved performance of this detection technique for the PDF Reader suggests that its baseline characteristics are more stable given the less varied range of inputs it handles compared to IE.

6 Analysis of Evasion Strategies

In general, anomaly-based intrusion detection approaches, such as ours, are susceptible to *mimicry* attacks. To evade detection, a sophisticated adversary with sufficient information about the anomaly detection models can modify her malware into an equivalent form that exhibits similar baseline architectural and microarchitectural characteristics as the normal programs. In this section, we examine the degree of freedom an adversary has in crafting a mimicry attack and how it impacts the detection efficacy of our models.

Adversary Assumptions. We assume the adversary (a) knows all about the target program such as the version and OS to be run on, and (b) is able to gather similar HPC measurements for the targeted program to approximate its baseline characteristics. (c) She also knows the way the events are modeled, but *not* the exact events used. We highlight three ways the adversary can change her attack while retaining the original attack semantics.

Assumption (c) is realistic, given the hundreds of possible events that can be monitored on a modern processor. While she may uncover the manner the events are modeled, it is difficult to pinpoint the exact subset of four events used given the numerous possible combinations of subsets. Furthermore, even if the entire event list that can be monitored is available, there may still exist some events (such as events monitored by the power management units) that are not publicly available. Nonetheless, to describe attacks 1 and 2, we optimistically assume the adversary has full knowledge of all the events that are used in the models.

Attack 1: Padding. The first approach is to pad the original shellcode code sequences with "no-op" (no effect) instructions with a sufficient number so that the events manifested by the shellcode match that of the baseline execution of the program. These no-op instructions should modify the measurements for all the events monitored, in tandem, to a range acceptable to the models.

The adversary needs to know the events used by the model *a priori*, in order to exert an influence over the relevant events. We first explore feasibility of such a mimicry approach by analyzing the *Stage1* shellcode under the detection model of event set *AM-1*. After studying the true positive samples, we observe that the event characteristics exhibited by the shellcode are due to the unusually low counts of the four events modeled. As we re-craft the shellcode at the assembly code level to achieve the mimicry effect, we note three difficulties.

(1) **Multi-instruction No-ops:** Some microarchitectural events require more than one instruction to effect a change. For example, to raise the MISP_RET counts, sequences of RET code need to be crafted in a specific order. Insertion of no-ops must be added in multi-instruction segments.

(2) **Event Co-dependence:** To maintain the original shellcode semantics, certain registers need to be saved and subsequently restored. These operations constitute STORE /LOAD μ-operations and can inadvertently affect both STORE and LOAD events. Thus we are rarely able to craft no-op code segments to modify each event independently. For instance, among the events in *AM-1*, only the no-op instruction segment for STORE can be crafted to affect it independently. Event co-dependence makes adversarial control of values of individual events challenging.

(3) **No-op Insertion Position:** Insertion position of the no-op instruction segments can be critical to achieve the desired mimicry effect. We notice the use of several loops within the shellcode. If even one no-op segment is inserted into the loops, that results in a huge artificial increase in certain event types, consequently making that code execution look more malicious than usual.

Next, we examine the impact of such mimicry efforts on the detection performance. We pad the *Stage1* shellcode at random positions (avoiding the loops) with increasing number of each crafted no-op instruction segment and repeated the detection experiments. In Figure 7 (*Left*), we plot the box-and-whisker plots of the anomaly scores observed from the samples with varying numbers of injected no-op code. In general, the anomaly scores become less anomalous with the padding, until after a tipping point where inserting too many no-ops reverses mimicry effect. In the same vein, we observe in Figure 7 (*Right*) that the detection AUC scores

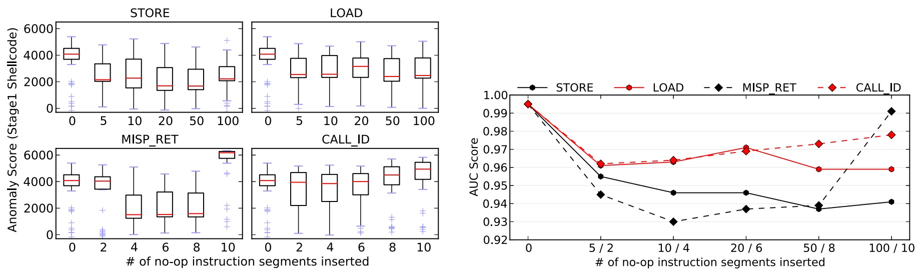

Fig. 7. Impact of inserting no-op segments on: **(Left)** The anomaly scores of *Stage1* shellcode and **(Right)** The detection efficacy of *Stage1* shellcode

decrease as the samples appear more normal. For the worst case, the detection performance suffers by up to 6.5% just by inserting *only* the CALL_ID no-ops. We do not study combining the no-ops for different events, but we believe it should deteriorate the detection performance further.

Attack 2: Substitution. Instead of padding no-ops into original attack code sequences, the adversary can replace her code sequences with equivalent variants using code obfuscation techniques, common in metamorphic malware [1]. Like the former attack, this also requires that she knows the events used by the models *a priori*. To conduct this attack, she must first craft or generate equivalent code variants of code sequences in her exploits, and profile the event characteristics of each variant. She can adopt a greedy strategy by iteratively substituting parts of her attack code with the equivalent variants, measuring the HPC events of the shellcode and ditching those variants that exhibit characteristics not acceptable to the models. However, while this greedy approach will terminate, it warrants further examination as to whether the resulting shellcode modifications suffice to evade the models. We argue that this kind of shellcode re-design is hard and will substantially raise the bar for exploit writers.

Attack 3: Grafting. This attack requires either inserting benign code from the target program directly into the exploit code, or co-scheduling the exploit shellcode by calling benign functions (with no-op effects) within the exploit code. This attack somewhat *grafts* its malicious code execution with the benign ones within the target program, thus relieving the need for the knowledge of the events that are modeled. If done correctly, it can exhibit very similar characteristics as the benign code it grafts itself to. As such this represents the most powerful attack against our detection approach.

While we acknowledge that we have not crafted this form of attack in our study, we believe that it is extremely challenging to craft such a grafting attack due to the operational constraints on the exploit and shellcode, described in Section 2. (1) Inserting sufficient benign code into the shellcode may exceed the vulnerability-specific size limits and cause the exploit to fail. (2) To use benign functions for the grafting attacks, these functions have to be carefully identified and inserted so that they execute sufficiently to mimic the normal program behavior and yet not interfere with the execution of the original shellcode. (3) The grafted code must not unduly increase the execution time of the entire exploit.

6.1 Defenses

Unlike past anomaly-based detection systems that detect deviations based on the syntactic/semantic structure and code behavior of the malware shellcode, our approach focuses on the architectural and microarchitectural side-effects manifested through the code execution of the malware shellcode. While the adversary has complete freedom in crafting her attack instruction sequences to evade the former systems, she cannot directly modify the events exhibited by her attack code to evade our detection approach. To conduct a mimicry attack here, she has to carefully "massage" her attack code to manifest a combination of event behaviors that are accepted as benign/normal under our models. This second-order degree of control over the event characteristics of the shellcode adds difficulty to the adversary's evasion efforts. On top of this, we discuss further potential defense strategies to mitigate the impact of the mimicry attacks.

Randomization. Introducing secret randomizations into the models has been used to strengthen robustness against mimicry attacks in anomaly-based detection systems [26]. In our context, we can randomize the events used in the models by training multiple models using different subsets of the shortlisted events. We can also randomize the choice of model to utilize over time. Another degree of randomization is to change the number of consecutive time-epoch samples to use for each sample for the temporal models. In this manner, the adversary does not know which model is used during the execution of her attack shellcode. For her exploit to be portable and functional on a wide range of targets, she has to modify her shellcode using the no-op padding and instruction substitution mimicry attacks for a wider range of events (and not just the current four events).

Multiplexing. At the cost of higher sampling overhead, we can choose to sample at a finer sampling granularity and measure more events (instead of the current four) by multiplexing the monitoring – we can approximate the simultaneous monitoring of 8 events across two time epochs by monitoring 4 events in one and another 4 in the other. This increases to the input dimensionality used in the models, making it harder for the adversary to make all the increased number of monitored event measurements appear non-anomalous.

Defense-in-depth. Consider a defense-in-depth approach, where this malware anomaly detector using HPC manifestations is deployed with existing anomaly-based detectors monitoring for other features of the malware, such as its syntactic and semantic structure [26,12,13] and its execution behavior at system-call level [22,6,15,20] and function level [17]. In such a setting, in order for a successful attack, an adversary is then forced to shape her attack code to conform to normalcy for each anomaly detection model. An open area of research remains in quantifying this multiplicative level of security afforded by the combined use of these HPC models with existing defenses, *i.e.* examining the difficulty in shaping the malware shellcode to evade detectors using statistical and behavioral software features, while simultaneously not exhibiting any anomalous HPC event characteristics during execution.

7 Related Work

The use of low-level hardware features for malware detection (instead of software ones) is a recent development. Demme *et al.* demonstrate the feasibility of misuse-based detection of Android malware programs using microarchitectural features [3]. While they model microarchitectural signatures of malware programs, we build baseline microarchitectural models of benign programs we are protecting and detect deviations caused by a potentially wider range of malware (even ones that are previously unobserved). Another key distinction is that we are detecting malware shellcode execution of an exploit within the context of the victim program during the act of exploitation; they target Android malware as whole programs. After infiltrating the system via an exploit, the malware can be made stealthier by installing into peripherals, or by infecting other benign programs. Stewin *et al.* propose detecting the former by flagging additional memory bus accesses made by the malware [23]. Malone *et al.* examine detecting the latter form of malicious static and dynamic program modification by modeling the architectural characteristics of benign programs (and excluding the use of microarchitectural events) using linear regression models [14]. Another line of research shows that malware can be detected using side-channel power perturbations they induce in medical embedded devices [2], software-defined radios [7] and mobile phones [11]. However, Hoffman *et al.* show that the use of such power consumption models can be very susceptible to noise, especially in a device with such widely varied use as the modern smartphone [8].

Besides HPCs, several works have leveraged other hardware facilities on modern processors to monitor branch addresses efficiently to thwart classes of exploitation techniques. kBouncer uses the Last Branch Recording (LBR) facility to monitor for runtime behavior of indirect branch instructions during the invocation of Windows API for the prevention of ROP exploits [16]. To enforce control flow integrity, CFIMon [28] and Eunomia [29] leverage the Branch Trace Store (BTS) to obtain branch source and target addresses to check for unseen pairs from a pre-identified database of legitimate branch pairs. Unlike our approach to detecting malware, these works are designed to prevent exploitation in the first place, and are orthogonal to our anomaly detection approach.

8 Conclusions

This work introduces the novel use of hardware-supported lower-level microarchitectural features to the anomaly-based detection of malware exploits. This represents the first work to examine the feasibility and limits of using unsupervised learning on microarchitectural features from HPCs to detect malware. We demonstrate that the dynamic execution of commonly attacked programs can be efficiently characterized with minimal features – the stream of event measurements easily accessible from the HPC, and used to detect lower-level perturbations caused by malware exploits to the baseline characteristics of benign programs. Unlike its misuse-based counterparts previously proposed, this anomaly-based detection approach can detect a wider range of malware, even novel ones. This work can thus be used in concert with its misuse-based counterparts to better security. Further, in modeling a class of potential mimicry attacks against our detector, we show that it can be challenging for an adversary to precisely control these hardware features to conduct an evasion attack.

Acknowledgements. We thank anonymous reviewers for feedback on this work. This work is supported by grants FA 865011C7190, FA 87501020253, CCF/SaTC 1054844 and a fellowship from the Alfred P. Sloan Foundation. Opinions, findings, conclusions and recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the US Government or commercial entities.

References

1. Borello, J.M., Mé, L.: Code obfuscation techniques for metamorphic viruses. Journal in Computer Virology 4(3), 211–220 (2008)
2. Clark, S.S., Ransford, B., Rahmati, A., Guineau, S., Sorber, J., Fu, K., Xu, W.: WattsUpDoc: Power Side Channels to Nonintrusively Discover Untargeted Malware on Embedded Medical Devices. In: USENIX Workshop on Health Information Technologies (August 2013)
3. Demme, J., Maycock, M., Schmitz, J., Tang, A., Waksman, A., Sethumadhavan, S., Stolfo, S.: On the feasibility of online malware detection with performance counters. In: Proceedings of the 40th Annual International Symposium on Computer Architecture, ISCA 2013, pp. 559–570. ACM, New York (2013)
4. Duda, R.O., Hart, P.E., Stork, D.G.: Pattern Classification. John Wiley & Sons, New York (2001), J. Classif. 24(2), 305–307, pp. xx + 654 (2007)
5. Fewer, S.: Reflective DLL injection (October 2008), `http://www.harmonysecurity.com/files/HS-P005_ReflectiveDllInjection.pdf`
6. Forrest, S., Hofmeyr, S.A., Somayaji, A., Longstaff, T.A.: A sense of self for unix processes. In: 1996 IEEE Symposium on Security and Privacy, pp. 120–128 (1996)
7. Gonzalez, C.R.A., Reed, J.H.: Detecting unauthorized software execution in sdr using power fingerprinting. In: Military Communications Conference, MILCOM 2010, pp. 2211–2216. IEEE (2010)
8. Hoffmann, J., Neumann, S., Holz, T.: Mobile Malware Detection Based on Energy Fingerprints A Dead End? In: Stolfo, S.J., Stavrou, A., Wright, C.V. (eds.) RAID 2013. LNCS, vol. 8145, pp. 348–368. Springer, Heidelberg (2013)
9. Hoste, K., Eeckhout, L.: Comparing Benchmarks Using Key Microarchitecture-Independent Characteristics. In: 2006 IEEE International Symposium on Workload Characterization, pp. 83–92. IEEE (October 2006)
10. Kayaalp, M., Schmitt, T., Nomani, J., Ponomarev, D., Abu-Ghazaleh, N.B.: SCRAP: Architecture for signature-based protection from Code Reuse Attacks. In: HPCA, pp. 258–269 (2013)
11. Kim, H., Smith, J., Shin, K.G.: Detecting energy-greedy anomalies and mobile malware variants. In: Proceedings of the 6th International Conference on Mobile Systems, Applications, and services. pp. 239–252. ACM (2008)
12. Kong, D., Tian, D., Liu, P., Wu, D.: SA3: Automatic semantic aware attribution analysis of remote exploits. In: Security and Privacy in Communication Networks, pp. 190–208. Springer (2012)

13. Mahoney, M.V.: Network traffic anomaly detection based on packet bytes. In: Proceedings of the 2003 ACM Symposium on Applied Computing, pp. 346–350 (2003)
14. Malone, C., Zahran, M., Karri, R.: 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 2011, pp. 71–76. ACM (2011)
15. Marceau, C.: Characterizing the behavior of a program using multiple-length n-grams. In: Proceedings of the 2000 Workshop on New Security Paradigms, pp. 101–110. ACM (2001)
16. Pappas, V., Polychronakis, M., Keromytis, A.D.: Transparent ROP exploit mitigation using indirect branch tracing. In: Proceedings of the 22nd USENIX Conference on Security, SEC 2013, pp. 447–462. USENIX Association, Berkeley (2013)
17. Peisert, S., Bishop, M., Karin, S., Marzullo, K.: Analysis of computer intrusions using sequences of function calls. IEEE Transactions on Dependable and Secure Computing 4(2), 137–150 (2007)
18. Polychronakis, M., Anagnostakis, K.G., Markatos, E.P.: Emulation-based detection of non-self-contained polymorphic shellcode. In: Kruegel, C., Lippmann, R., Clark, A. (eds.) RAID 2007. LNCS, vol. 4637, pp. 87–106. Springer, Heidelberg (2007)
19. Polychronakis, M., Anagnostakis, K.G., Markatos, E.P.: Comprehensive shellcode detection using runtime heuristics. In: Proceedings of the 26th Annual Computer Security Applications Conference, pp. 287–296. ACM (2010)
20. Sekar, R., Bendre, M., Dhurjati, D., Bollineni, P.: A fast automaton-based method for detecting anomalous program behaviors. In: Proceedings of the 2001 IEEE Symposium on Security and Privacy, S&P 2001, pp. 144–155. IEEE (2001)
21. Shen, K., Zhong, M., Dwarkadas, S., Li, C., Stewart, C., Zhang, X.: Hardware counter driven on-the-fly request signatures. In: Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XIII, pp. 189–200. ACM, New York (2008)
22. Somayaji, A., Forrest, S.: Automated response using system-call delays. In: Proceedings of the 9th USENIX Security Symposium, vol. 70 (2000)
23. Stewin, P.: A primitive for revealing stealthy peripheral-based attacks on the computing platforms main memory. In: Stolfo, S.J., Stavrou, A., Wright, C.V. (eds.) RAID 2013. LNCS, vol. 8145, pp. 1–20. Springer, Heidelberg (2013)
24. Szor, P.: The art of computer virus research and defense. Pearson Education (2005)
25. TrendMicro: The crimeware evolution (research whitepaper) (2012)
26. Wang, K., Parekh, J.J., Stolfo, S.J.: Anagram: A content anomaly detector resistant to mimicry attack. In: Zamboni, D., Kruegel, C. (eds.) RAID 2006. LNCS, vol. 4219, pp. 226–248. Springer, Heidelberg (2006)
27. Wang, X., Karri, R.: NumChecker: Detecting kernel control-flow modifying rootkits by using hardware performance counters. In: Proceedings of the 50th Annual Design Automation Conference, DAC 2013, pp. 79:1–79:7. ACM, NY (2013)
28. Xia, Y., Liu, Y., Chen, H., Zang, B.: CFIMon: Detecting violation of control flow integrity using performance counters. In: Proceedings of the 2012 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2012, pp. 1–12. IEEE Computer Society, Washington, DC (2012)
29. Yuan, L., Xing, W., Chen, H., Zang, B.: Security breaches as PMU deviation: detecting and identifying security attacks using performance counters. In: APSys, p. 6 (2011)
30. Garfinkel, T., Rosenblum, M.: A Virtual Machine Introspection Based Architecture for Intrusion Detection. In: NDSS, vol. 3 (2003)