Hardware-Assisted Rootkit Detection via On-line Statistical Fingerprinting of Process Execution

Liwei Zhou and Yiorgos Makris
Electrical and Computer Engineering Department, The University of Texas at Dallas, Richardson, TX 75080, USA
E-mail: {lxz100320, gxm112130}@utdallas.edu

***Abstract*—Kernel rootkits generally attempt to maliciously tamper kernel objects and surreptitiously distort program execution flow. Herein, we introduce a hardware-assisted hierarchical on-line system which detects such kernel rootkits by identifying deviation of dynamic intra-process execution profiles based on architecture-level semantics captured directly in hardware. The underlying key insight is that, in order to take effect, malicious manipulation of kernel objects must distort the execution flow of benign processes, thereby leaving abnormal traces in architecture-level semantics. While traditional detection methods rely on software modules to collect such traces, their implementations are susceptible to being compromised through software attacks. In contrast, our detection system maintains immunity to software attacks by resorting to hardware for trace collection. The proposed method is demonstrated on a Linux-based operating system running on a 32-bit x86 architecture, implemented in Simics. Experimental results, using real-world kernel rootkits, corroborate the effectiveness of this method, while a predictive 45nm PDK is used to evaluate hardware overhead.**

I. Introduction

As the number of transactions performed on-line and the amount of private information that is stored and communicated between electronic devices increases, millions of malicious software (or *malware*) continue to emerge [1], leading to service disruptions and/or security breaches. As a result, the decades-long arms race between malware and defense mechanisms perpetuates and intensifies.

Although numerous malware detection mechanisms have been developed, whose preliminary experiments have shown favorable results, there remains one species of malware, i.e., kernel rootkits, whose detection is far from promising. In general, kernel rootkits have unrestricted access to operating system (OS) resources and attempt to tamper kernel objects and inject malicious code stealthily. Traditional malware detection methods seek to model program behavior and, therefore, train a 2-class classifier, in order to distinguish malicious processes from benign ones. In other words, these methods detect malware through *inter-process behavior deviation*. However, such a detection mechanism may fail under rootkit attack scenarios. For example, an attack can be launched by implanting a rootkit that injects malicious code in the original system call table. This results in distortion in the execution flow of existing processes, rather than creation of a new (covert) malware instance. In such cases, rootkit-infected processes, whose behavior deviates only slightly from their legitimate version due to malicious actions, may not be detected. Evidently, the escalated privileges and the implementation specifics of kernel rootkits makes them harder to detect using traditional malware detection strategies.

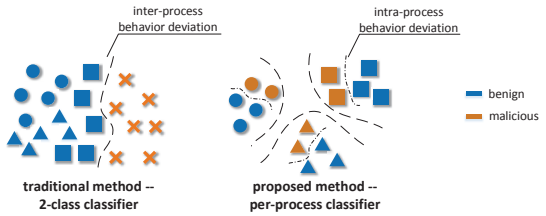

Fig. 1: Traditional vs. proposed method

Furthermore, most of the state-of-the-art malware detection methods were developed at the OS- or hypervisor-level [2]–[7]. OS-level methods benefit from semantic-rich information, e.g., process ID, file system objects, etc. Nevertheless, they are susceptible to software attacks launched from the same privilege domain. To address this limitation, hypervisor-level methods were proposed, since hypervisors operate with higher privileges. Unfortunately, the hypervisor itself can be the attack target, as several vulnerabilities and intrusion methods have been identified [8]. Consequently, software-based detection approaches may suffer the risk of corruption of the logged data or even disabling of the detection system.

To address the aforementioned limitations, we propose a new rootkit detection mechanism, wherein a dynamic program execution profile is modeled individually for each process, using a machine learning approach, in order to identify whether a process is rootkit-infected. In other words, our detection mechanism relies on *intra-process behavior deviation*, as shown in Fig. 1. As a result, a more precise view of process execution profile is constructed at a finer granularity and, thus, even slight deviations of process execution flow incurred by kernel rootkits can be detected. Moreover, we explore the possibility of a hardware-assisted solution, which relies on information obtained exclusively from the hardware. Accordingly, data traces collected through hardware are expected to be immune to any software tampering. Our idea is demonstrated through execution of both legitimate benchmarks and real-world kernel rootkits on an x86 architecture running Linux OS.

The rest of the paper is structured as follows. In Section II, we briefly discuss related work. The threat model considered is introduced in Section III. The proposed method is illustrated in Section IV, while hardware implementation details are provided in Section V. We evaluate our system and its overhead in Section VI, while potential limitations are discussed in Section VII. Conclusions are drawn in Section VIII.

978-3-9819263-0-9/DATE18/©2018 EDAA

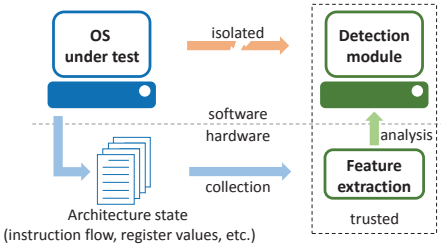

Fig. 2: System architecture of proposed method

II. Related Work

Based on their object of interest, state-of-the-art hardware-assisted malware detection or rootkit detection methods can be categorized into *data-centric* and *program-centric*. The former typically generate and validate static signatures for specific objects, in order to detect malicious events which may jeopardize data integrity. For example, detecting malware through Control Flow Integrity (CFI), which seeks to identify illegitimate redirection of program control flow, has been a popular data-centric solution. In one such work, a CFI module is built in hardware, wherein signatures of basic blocks are evaluated through Hamming distance, in order to perform intrusion detection [9]. In a similar kernel rootkit detection solution [10], a low-cost system call routine fingerprinting method which employs custom hardware components, such as a Multiple Input Signature Register (MISR) and a Bloom filter, to generate and validate fingerprints of system call execution, has been proposed. These methods generally lead to promising detection efficiency. Nevertheless, such static signature-based methods require prior knowledge of the binary image of the objects which they focus on, and their effectiveness may be undermined when indirect jumps are involved.

On the other hand, program-centric approaches seek to leverage low-level information extracted directly from hardware in order to model dynamic program behavior and perform the targeted analysis. For instance, performance counters have been widely used to model program behavior through machine learning methods in order to perform malware detection or rootkit detection [11], [12]. Alternative approaches mine the instruction flow and collect other low-level architectural information, such as memory address references, instruction opcodes, etc., to perform a similar analysis [13]–[15]. These methods typically follow the general strategy of traditional malware detection solutions, as introduced in Section I, and therefore, have limited effectiveness in rootkit detection or do not even address the problem at all.

III. Threat Model

In this section, we define the threat model considered in this work. Particularly, we aim at kernel rootkits which are assumed to (i) have full access to the OS memory image, and (ii) be able to make arbitrary modifications and execute malicious code in OS kernel space. As a result, the rootkits are able to hijack the control flow of arbitrary kernel services, e.g., system calls, and hook their malicious activities onto random benign processes. Furthermore, unlike previous malware or rootkit detection methods which require availability of known malware/rootkit samples during their program behavior modeling phase [7], [11], [13], we assume no prior knowledge of the kernel rootkits, i.e., the contaminated objects, the rootkit payload, or the binary image of the rootkits. In other words, we assume a *zero-day attack* scenario.

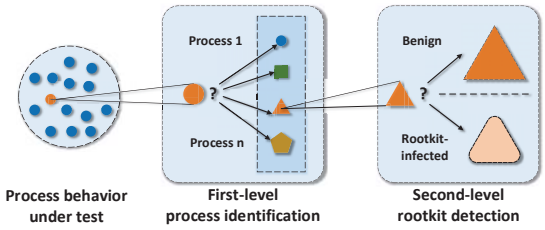

Fig. 3: Rootkit detection flow

IV. Design of the Proposed Method

A. System Architecture

A top-level view of the proposed rootkit detection method is shown in Fig. 2. Unlike software-based approaches, our method mines the architecture state and extracts relevant information exclusively from the hardware, in order to ensure integrity of the logged data. The collected data is then offloaded to a trusted software environment, which is isolated from the original OS, and wherein rootkit detection is performed by a machine learning entity which has been trained to model the intra-process behavior. Since this collection-to-analysis path does not interfere with the original OS execution and can be performed in parallel, our method is non-intrusive and incurs no runtime overhead.

The actual rootkit detection is performed through a hierarchical mechanism, as shown in Fig. 3. When a new process sample arrives, a first-level process identification is applied to identify what process class it belongs to. After that, a second-level rootkit detection is performed on the corresponding process class independently, in order to investigate whether a process sample is truly benign or rootkit-infected.

To perform the entire detection flow, we have to address several challenges, as explained below: (i) **Process identifier**: In order to perform rootkit detection at the process level through a hardware-assisted method, we need to bridge the semantic gap between architecture-level logged information and the actual processes. (ii) **Program behavior**: Descriptive features need to be extracted from information available in the hardware in order to model program behavior. (iii) **Machine learning**: Using the collected data, an appropriate machine learning method is required in order to perform the proposed hierarchical analysis.

B. Process Identifier

In modern OSs, due to the virtual memory concept, each process has its own dedicated address space, which maps resources used by the process into physical memory. This mapping is facilitated by the translation between virtual address and physical address, maintained by a per-process page table. In x86, the base address of this table is stored in a control

TABLE I. Summary of feature set

Type	Description
DP[1-24]	counts of 3 types of data dependencies on 4 general-purpose registers in 2 OS modes
BR[25-27]	counts of 3 types of branches (i.e., within and across the 2 OS modes)

register, `CR3`. Changes of the `CR3` value perfectly match the events of process creation, switching and termination [16]. As a result, we use the `CR3` value as the identifier of a process.

C. Feature Extraction

While the behavior of a program can be explained through execution of its instruction flow in a microprocessor, it is impractical to log the entire instruction flow in hardware. As a result, hardware-assisted malware/rootkit detection methods generally leverage architecture-level information, which indirectly reflects data and control transfer flow, in order to model program behavior. Along these lines, our method seeks to model program behavior through hardware events representing change of microprocessor state, including register usage, program control flow redirection, OS operation state, etc. During rootkit execution, these hardware events will deviate from those occurring during a benign execution path, thereby leaving traces that can be used for rootkit detection. In particular, our method interpretes the program data/control transfer flow through hardware events involving data dependencies between registers, branches in program execution flow, and OS privilege transition.

Data dependencies exist when an instruction involves target or source operands which are also referenced by preceding instructions. Such dependencies need to be resolved prior to instruction execution, in order to preserve correct program functionality. Three types of data dependencies exist: (i) True dependency occurs when an instruction reads a register being written by a preceding instruction. (ii) Anti-dependency occurs when an instruction writes a register being read by a preceding instruction. (iii) Output dependency occurs when an instruction writes a register being written by a preceding instruction.

In x86, four general purpose registers, i.e., `eax`, `ebx`, `ecx`, and `edx`, are most frequently used. Our method collects counts of the three types of dependencies on each of the registers as our data dependency-related features. Furthermore, instructions can operate in user mode or kernel mode in a modern OS. Data dependency statistics are collected separately for these two modes, leading to a deeper understanding of how a process operates in its user space and in kernel space. Ultimately, for each `CR3` value representing a process, 24 data dependency-related features are collected.

Regarding branches in program execution flow, we consider 3 types of branches, including intra-user, user-kernel and intra-kernel branches. Intra-user branches involve jumps between user-space instructions, capturing the functionality of a program in user mode. User-kernel branches, on the other hand, involve transition between user and kernel mode. Such transition may occur due to either software interrupts, which are launched actively by program execution, or hardware interrupts, which are asynchronous with the program execution flow. Since we aim at modeling program behavior with minimal impact on the underlying environment, only branches introduced by software interrupts, launched by programs explicitly through `SYSCALL` or `INT` instructions, are considered. Finally, similar to intra-user branches, intra-kernel branches are collected accordingly. Table I summarizes the features considered in this work.

D. Rootkit Detection

Upon extracting the aforementioned features, our detection mechanism employs machine learning to perform a hierarchical analysis, i.e., a first-level process identification and a second-level rootkit detection.

1) Process Identification: The process identification method employs multi-class classification algorithms, where each class corresponds to a single process. We experimented with three classifiers of varying complexity and performance, namely K-Nearest Neighbors (KNN), Support Vector Machine (SVM) and Artificial Neural Network (ANN).

KNN is a non-parametric classification algorithm which classifies samples based on spatial relationship in their feature space. It computes the *k* nearest neighbors of a sample using Euclidean distance and assigns the sample to a class based on majority voting among these neighbors. SVM, on the other hand, generates a hyperplane which separates the transformed feature space into labeled sub-spaces, while ensuring maximal separation among them. ANN exploits a multi-layer structure, where each layer consists of multiple nodes, i.e., *neurons*, which are interconnected with nodes in adjacent layers. Through stacked layers, ANN maps the original inputs, via an activation function on each neuron, to a final labeled space, accomplishing the classification. In our implementation, we used KNN from the Matlab library, SVM from the LIBSVM library [17] and ANN from Keras [18].

2) Rootkit Detection: After identifying the process class that a sample belongs to, a second-level rootkit detection is performed. To this end, we employ an outlier detection method, wherein an outlier indicates that the process behavior has been compromised and a rootkit is detected. Specifically, since the probability distribution of the feature space of processes is unknown, we use Kernel Density Estimation (KDE) which can handle unknown input probability distributions.

KDE evaluates the probability density of the samples under test using an adaptive kernel estimator and identifies outliers outside the probability distribution. Outlier detection is then performed as follows. Given a benign-sample matrix X including n samples, each of which has d features, its kernel estimator is defined by:

$$\tilde{f}(x) = \frac{1}{nh^d} \sum_{i=1}^{n} K(\frac{1}{h}(x - X_i)) \quad (1)$$

where K is the kernel function and h is an adjustable smoothing parameter called bandwidth. The kernel we use herein is the Epanechnikov kernel:

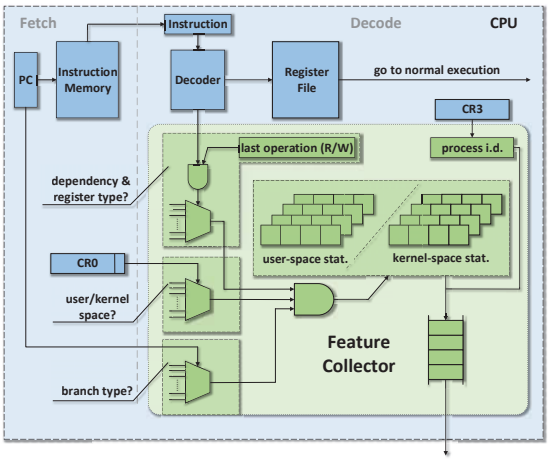

Fig. 4: Hardware implementation of feature extraction

$$K_e(t) = \begin{cases} \frac{1}{2}c_d^{-1}(d+2)(1-t^Tt), & t^Tt < 1 \\ 0, & otherwise \end{cases} \quad (2)$$

where $c_d^{-1} = 2\pi^{d/2}/(d \cdot \Gamma(d/2))$ is the volume of the unit d-dimensional sphere [19]. A rule-of-thumb choice of h is:

$$h = \left\{8c_d^{-1}(d+4)(2\sqrt{\pi})^d\right\}^{1/(d+4)} n^{-1/(d+4)} \quad (3)$$

For a sample-under-test matrix Y including m samples, each of which has d features, its adaptive kernel estimator is:

$$\hat{f}(y) = \frac{1}{n}\sum_{i=1}^{m} \frac{1}{(h \cdot \lambda_i)^d} K(\frac{1}{h \cdot \lambda_i}(x - Y_i)) \quad (4)$$

where the local bandwidth scalars λ_i are defined by:

$$\lambda_i = \left\{\tilde{f}(X_i)/g\right\}^{-\alpha} \quad (5)$$

$\tilde{f}(X_i)$ is a pilot density estimate calculated in (1) with h defined in (3). g is the geometric mean given by:

$$\log g = n^{-1}\sum_{i=1}^{n} \log \tilde{f}(X_i) \quad (6)$$

while α is a sensitivity parameter $\in [0, 1]$. After obtaining the probability density estimate for the samples under test, a threshold is set to filter outliers. Probability lower than the threshold indicates a rootkit-infected process, while probability greater than the threshold indicates a benign process. Parameters of the outlier detection model, such as h, α and the threshold are tuned for each process class individually, in order to optimize detection performance.

V. Hardware Implementation

As mentioned earlier, our feature extraction is performed directly in hardware to eliminate the possibility of software tampering. The actual implementation, as shown in Fig. 4, employs a custom hardware component, i.e., feature collector, which is deeply coupled with the CPU and collects the process identifier as well as the features used for process behavior modeling, based on the microprocessor state.

TABLE II. Summary of rootkit samples

Rootkit	Targeted system call	Rootkit	Targeted system call
maKit	write, open, read	**lkm-syscall**	open, close
suterusu	ioctl, read, write	**hook-syscall**	mkdir
syscall-hooker	read, write	**simple-rootkit**	read
hijack-syscall	open	**Diamorphine**	getdents, kill

In order to collect process identifiers, the feature collector captures the `CR3` register value whenever a value update is encountered. As mentioned in Section IV-C, user and kernel mode features are collected separately. To determine which mode an instruction operates in, we leverage the design convention of control register `CR0` in x86. The least significant bit of `CR0` register, or `PE` bit, indicates which mode the underlying system operates in, with '1' meaning kernel mode and '0' meaning user mode. Therefore, the feature collector is wired to the `PE` bit to split program instructions into user-space and kernel-space instructions, respectively.

Statistics of data dependencies can, then, be generated for instructions in different modes. To this end, we make use of the built-in decoder in the microprocessor to derive the read/write operations on the 4 general purpose registers. A temporary register is associated with each of the general purpose registers, maintaining the `READ` or `WRITE` tag of its last access. For every new access on a general purpose register, its current operation is compared with its last operation, thereby identifying dependency pairs. Accordingly, a counter corresponding to the specific dependency type is incremented by '1' while the temporary register is updated. Collecting the 3 types of branch statistics, meanwhile, is more straightforward. The feature collector continuously monitors the program counter and instruction operators, in order to detect the occurrence of the branch events defined in Section IV-C, and updates the corresponding counters.

VI. Experimental Results

In this Section, we evaluate the efficacy of our proposed method in accurately classifying processes and successfully detecting rootkits. Additionally, we evaluate the area/power overhead and logging bandwidth required by the hardware implementation of this method. The experiments were performed in Simics. Therein, an x86 machine is simulated, configured with a single Intel Pentium 4 core running at 2GHz. A minimum installation Ubuntu server that embeds a Linux 3.8 kernel is loaded on the simulated hardware platform. As benign workload, we used Mibench [20], a free commercially representative benchmark suite, which contains tens of applications. As rootkit samples, similar to the dataset used in [7], we experimented with real-world Linux rootkits summarized in Table II, which hijack arbitrary system call service routines to perform denial-of-service attack, file/process hiding, key logging, etc. Implementations of rootkit samples have been elaborated to create more variants, since only a generalized template is provided in the original version.

TABLE III. Process identification accuracy

	KNN	SVM	ANN
average	100/**99.85**%	99.90/**99.87**%	99.89/**99.87**%
bf	100/100%	100/99.71%	100/100%
qsort	100/100%	99.34/99.73%	100/100%
patricia	100/100%	99.28/99.72%	100/100%
toast	100/100%	100/100%	100/100%
untoast	100/100%	100/100%	100/100%
susan	100/100%	100/100%	100/100%
dijkstra	100/100%	100/100%	100/100%
sha	100/100%	100/100%	100/100%
crc	100/100%	100/100%	100/100%
search	100/98.26%	100/100%	100/98.26%
tiff2rgba	100/100%	99.34/99.10%	100/100%
tiff2bw	100/100%	100/99.17%	98.50/99.69%
tiffmedian	100/100%	100/100%	100/100%
basicmath	100/100%	100/100%	100/100%
rawcaudio	100/100%	100/100%	100/100%
rawdaudio	100/100%	100/100%	100/100%
fft	100/98.83%	100/100%	99.29/99.41%
cjpeg	100/100%	100/100%	100/100%
djpeg	100/100%	100/100%	100/100%
pgp	100/100%	100/100%	100/100%

A. Process Identification

To evaluate the accuracy of our method in process identification, the Mibench suite was executed repeatedly, with each application invoked with various valid arguments or in the background (& option). Rootkit-infected samples are created by executing Mibench after enabling different rootkits. In total, we collected approximately 20000 benign as well as 10000 rootkit-infected samples, evenly for 20 process classes. The benign dataset was split in half for training and testing, while the rootkit-infected dataset was used only in testing.

The process identification results using KNN, SVM and ANN are shown in Table III. The numbers on the left of the slash represent identification accuracy using the testing set excluding rootkit-infected samples, while the numbers on the right side represent the case including these samples. As may be observed, there is no significant difference in the results between the two cases, indicating that rootkit-infected processes did not incur higher misclassification, *even though the training set contained only benign processes*. This observation supports our conjecture that the rootkit-infected process behavior may not be distinguishable from its benign instances through inter-process behavior deviation. Furthermore, all three classifiers performed well in identifying processes (including rootkit-infected processes), reaching an average accuracy of 99.85%, 99.87%, and 99.87% respectively. This provides solid ground for the next-level rootkit detection.

B. Rootkit Detection

Effectiveness of our method in rootkit detection was evaluated separately for each process class. Benign samples in each class were split in half for training and testing, while rootkit-infected samples were only used for testing. The parameters of the KDE algorithm were optimized independently for each class through cross-validation to maximize rootkit detection capability with minimal false alarms. During the optimization of KDE parameters, only a subset of the rootkit family under test was used, in order to avoid overfitting to the current rootkit dataset, as well as to ensure resilience of our detection model to zero-day rootkit samples.

TABLE IV. Per-process rootkit detection results

process class	FP rate	FN rate	process class	FP rate	FN rate
bf	1.41%	0%	**tiff2rgba**	1.33%	0%
qsort	0%	0%	**tiff2bw**	1.74%	0%
patricia	0.73%	0%	**tiffmedian**	0.98%	0%
toast	0%	0%	**basicmath**	0.75%	0%
untoast	0.67%	0%	**rawcaudio**	0%	0%
susan	0.49%	0%	**rawdaudio**	0%	0%
dijkstra	1.4%	0%	**fft**	0%	0%
sha	1.43%	0%	**cjpeg**	1.8%	0%
crc	0.69%	0%	**djpeg**	0%	0%
search	1.49%	0%	**pgp**	0%	0%

Table IV summarizes the per-class false positive (FP) (i.e., benign process identified as rootkit-infected) and false negative (FN) (i.e., rootkit-infected process identified as benign) rates. As may be observed, the worst FP rate is 1.8% and the average is 0.75%, while 0% FN rates are achieved for all process classes under test. Indeed, intra-process behavioral models describe process activities at a finer granularity, making rootkit-infected behavior distinguishable. We emphasize that our method outperforms the state-of-the-art hardware-assisted malware detection method, which achieves no significant result in rootkit detection [11]. Furthermore, compared with the software-based counterpart which achieves similarly promising results (i.e., 100% detection rate with low FP rate) [7], our method is inherently more secure since it extracts data in hardware through a custom component. Moreover, it incurs zero runtime overhead due to the non-intrusive collection-to-analysis path. In contrast, the software-based solution incurs a runtime overhead of approximately 3% [7].

C. Overhead

To evaluate the design overhead of the proposed method, we focus on (i) additional area and power overhead introduced by the feature collector, and (ii) the required data logging bandwidth. Herein, we evaluate the area/power overhead by synthesizing the design of the feature collector using a predictive 45nm Process Design Kit (PDK) [21], which results in area overhead of 649.98 μm^2 and power overhead of 1.9152 mW (at 2GHz). Compared to a 45nm Intel processor[1], the additional overhead incurred by the feature collector is negligible. Furthermore, we ran our workload multiple times to obtain an average estimation of the data logging rate,

[1] Specifications from http://ark.intel.com/products/35605

TABLE V. Design overhead of the proposed method

	area(μm^2)	**power**(mW)	**logging**(KB/s)
this method	649.98	1.9152	50.51
processor	107×10^6	65×10^3	N/A

resulting in a rate of 50.51KB/s. As a point of reference, the performance counter-based method in [11] requires bandwidth of a few hundred KB/s to perform similar analysis. Table V summarizes the design overhead of the proposed method.

VII. Discussion

A. Register Renaming/Reorder Buffer

Modern microprocessors exploit techniques such as *register renaming* or *Reorder buffer (ROB)* to improve performance. Register renaming *renames* a register to an idle one when a writing request occurs, so that this operation can be executed before its preceding instruction which reads the same register. Similarly, a ROB leverages a register buffer as temporary storage to hold values of speculatively executed instructions. These techniques eliminate anti-dependencies as well as output dependencies and enable out-of-order and speculative program execution. However, they do not affect negatively the proposed method effectiveness, as our feature collector investigates data dependencies when instructions are fetched and decoded in-order, before these techniques are applied.

B. On-chip Detection Solution

Hardware-based malware detection can be implemented on-chip [13], [14]. Our method, however, implements only the feature extraction mechanism in hardware and exports the logged data to a trusted software environment to perform off-chip analysis. In fact, there is a trade-off between on-chip and off-chip solutions. The former generally benefit from prompt reaction to malicious events as compared with the latter; implementing the analysis module on chip, however, increases the design complexity and overhead. Furthermore, when the underlying OS or applications release an update, the configuration of the on-chip analysis module must be updated accordingly, which is not at all straightforward. In contrast, an off-chip analysis module is slower in responding but more flexible, as it can be updated while the on-chip logging component remains unchanged.

VIII. Conclusion

We introduced a hardware-assisted infrastructure for performing on-line rootkit detection. Compared with software-based solutions, whose effectiveness may be undermined by software attacks, we extract data of interest directly from the hardware, making our system resistant to software tampering. Moreover, unlike traditional malware or rootkit detection methods, which assume prior knowledge of malware/rootkit dataset and perform 2-class classification analysis based on inter-process behavior deviation, this method assumes a zero-day attack scenario and proposes a new hierarchical detection mechanism, leveraging intra-process behavior deviation and outlier detection. An incarnation of this idea, which models per-process behavior using data-dependencies, branch statistics and privilege transition, based on which rootkit detection can be performed, was described herein. Experimental results using the Mibench suite with real-world kernel rootkits revealed that almost perfect detection accuracy with very low false positive rate can be achieved. The required logging bandwidth of our method is 50.51KB/s while its hardware cost is negligible compared to the size of a modern microprocessor.

References

[1] McAfee Labs. Threats report. https://www.mcafee.com/au/resources/reports/rp-quarterly-threats-jun-2017.pdf, 2017.

[2] A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda. Accessminer: Using system-centric models for malware protection. In *17th ACM conf. on CCS*, pages 399–412, 2010.

[3] X. Ge, N. Talele, M. Payer, and T. Jaeger. Fine-grained control-flow integrity for kernel software. In *IEEE European Symp. on S & P*, 2016.

[4] L. Zomlot, S. Chandran, D. Caraea, and X. Ou. Aiding intrusion analysis using machine learning. In *12th Intl. Conf. on Machine Learning and Applications*, 2013.

[5] Y. Fu and Z. Lin. Space traveling across VM: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection. In *IEEE Symp. on S & P*, pages 586–600, 2012.

[6] S. Krishnan, K. Snow, and F. Monrose. Trail of bytes: New techniques for supporting data provenance and limiting privacy breaches. *IEEE Trans. on Information Forensics and Security*, 7(6):1876–1889, 2012.

[7] X. Wang and R. Karri. Reusing hardware performance counters to detect and identify kernel control-flow modifying rootkits. In *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, volume 35, pages 485–498, 2016.

[8] D. Perez-Botero, J. Szefer, and R. Lee. Characterizing hypervisor vulnerabilities in cloud computing servers. In *Intl. Workshop on Security in Cloud Computing*, pages 3–10, 2013.

[9] A. Kanuparthi, J. Rajendran, and R. Karri. Controlling your control flow graph. In *IEEE Intl. Symp. on HOST*, 2016.

[10] L. Zhou and Y. Makris. Hardware-based on-line intrusion detection via system call routine fingerprinting. In *DATE*, pages 1546–1551, 2017.

[11] J. Demme et al. On the feasibility of online malware detection with performance counters. In *40th Annual Intl. Symp. on Computer Architecture*, pages 559–570, 2013.

[12] A. Tang, S. Sethumadhavan, and S. J. Stolfo. Unsupervised anomaly-based malware detection using hardware features. In *Proc. of 17th Intl. Symp. RAID*, pages 109–129, 2014.

[13] 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 Intl. Symp. on HPCA*, pages 651–661, 2015.

[14] K. N. Khasawneh, M. Ozsoy, C. Donovick, N. B. Abu-Ghazaleh, and D. V. Ponomarev. Ensemble learning for low-level hardware-supported malware detection. In *18th Intl. Symp. on RAID*, pages 3–25, 2015.

[15] L. Zhou and Y. Makris. Hardware-based workload forensics: Process reconstruction via TLB monitoring. In *IEEE Intl. Symp. on HOST*, pages 167–172, 2016.

[16] S. Jones, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. Antfarm: Tracking processes in a virtual machine environment. In *Annual Conf. on USENIX*, pages 1–14, 2006.

[17] C. Chang and C. Lin. LIBSVM: A library for support vector machines. *ACM Trans. on Intelligent Systems and Technology*, 2:1–27, 2011.

[18] Keras. https://github.com/fchollet/keras, 2015.

[19] H.-G. Stratigopoulos, S. Mir, and A. Bounceur. Evaluation of analog/rf test measurements at the design stage. In *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 2009.

[20] M.R. Guthaus et al. Mibench: A free, commercially representative embedded benchmark suite. In *IEEE Intl. Workshop on Workload Characterization*, pages 3–14, 2001.

[21] J. Stine, I. Castellanos, M. Wood, J. Henson, and F. Love. FreePDK: An open-source variation-aware design kit. In *Proc. of the IEEE Intl. Conf. on Microelectronic Systems Education*, pages 173–174, 2007.