Hardware-based Workload Forensics and Malware Detection in Microprocessors

Liwei Zhou and Yiorgos Makris
Electrical Engineering Department, The University of Texas at Dallas
Richardson, TX 75080, USA

***Abstract*—We investigate the possibility of performing workload forensics and/or malware detection in microprocessors through exclusively hardware-based methodologies. Specifically, we first introduce a general architecture which a hardware-based forensics or malware detection method would need to follow, as well as the various processor-level information which could potentially be harnessed to ensure system security and/or integrity. In contrast to traditional forensics and/or malware detection methods implemented at the operating system (OS) and/or the hypervisor level, whose data logging and monitoring systems are vulnerable to spoofing attacks at the same level, moving implementation to hardware ensures immunity to such attacks. This work focuses on two recent incarnations of this general concept, illustrating the effectiveness of hardware-based forensics and/or malware detection. Several other recent methods related to this topic are also discussed. Experimental results corroborate that even a low-cost hardware implementation can facilitate highly successful forensics analysis and/or malware detection, while taking advantage of its innate immunity to software-based attacks.**

I. Introduction and Motivation

Over the last few decades, the prevalence of electronic devices has resulted in rapidly increasing amounts of private/sensitive information, such as personal details or trade secrets, being stored, processed and exchanged in electronic form. Unfortunately but inevitably, this has also lead to the emergence of hundreds of millions of malicious software[1], or *malware*, which seek to interfere with the underlying computer systems and to steal or disrupt such information, in order to benefit from such illegitimate access. As a result, developing a defense mechanism which is able to identify the events that led to the compromising of sensitive data, during or after such malicious acts occur, becomes indispensable.

Traditional methods to achieve this goal can be categorized into retroactive investigation, known as computer forensics, and active defense, known as malware/intrusion detection. The former performs *ex post facto* analysis on data of interest in order to reveal and/or reconstruct the events that occurred. The latter, on the other hand, detect and/or prevent the execution of potential threats in the underlying computer system in real time. In each of these categories, a large number of methods have been developed by industry and academia. Most of them focus at the operating system (OS) level, wherein they take advantage of the rich semantics at this level, such as OS-level objects, file system structure, etc. [1]–[3]. OS-level solutions, however, can be vulnerable to software attacks staged at the same level. Kernel rootkits, for example, may be used to hijack OS control flow so that it can spoof an OS-level logging/monitoring system and eliminate traces associated with malicious actions.

To address this limitation, hypervisor-level methods have also been proposed [4]–[6]. A hypervisor provides virtualization, thereby allowing multiple operating systems (guests) to run concurrently on a single physical machine, without intruding each other's context. A management core, designed to be isolated from the guest-OSs whose execution is facilitated by the hypervisor, may naturally provide ground for more secure forensics or malware detection solutions. Therefore, data collected by an investigation/defense system at the hypervisor level is generally immune to OS-level software attacks. Nevertheless, the hypervisor itself can be the target, as several vulnerabilities and intrusion methods have been identified in recent work [7]. As a result, similar attacks compromising integrity of the logged data of interest to conceal malicious events can also be staged at the hypervisor level.

Given the weakness of both OS-level and hypervisor-level approaches, recent research has explored the possibility of performing forensics analysis and/or malware detection through hardware-based methodologies [8]–[12]. Specifically, hardware-based approaches rely exclusively on data collected directly through the hardware, without the intervention of a hypervisor or an OS, whereby the logged information may be compromised. Accordingly, traces obtained from hardware are expected to be immune to software-based tampering. On the other hand, a hardware-based solution requires circuitry addition and modification in the microprocessor for identifying, extracting, and logging the relevant information. Therefore, judicious selection of information sufficient for fulfilling the targeted task becomes crucial. To this end, most of the current methods rely on processor-level information available through relatively low-cost extraction mechanisms, in order to ensure their practicality.

The remainder of the paper is structured as follows. In Section II, we introduce the general architecture which most hardware-based methodologies need to follow, as well as the potentially helpful information for performing forensics analysis and/or malware detection. Two incarnations illustrating the proposed concept, as well as their corresponding experimental results and design overhead, are introduced in Section III. Several other recent related works are presented in Section IV, while conclusions are drawn in Section V.

[1]http://www.mcafee.com/us/resources/reports/rp-quarterly-threats-sep-2016.pdf

2332-5674/16 $31.00 © 2016 IEEE
DOI 10.1109/MTV.2016.20

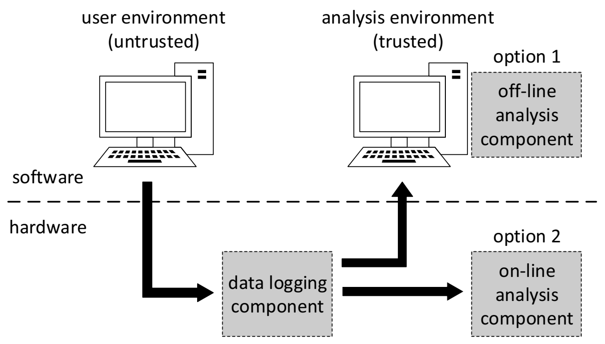

user environment (untrusted)
analysis environment (trusted)
option 1
off-line analysis component
software
hardware
option 2
data logging component
on-line analysis component

Fig. 1: High-level system architecture

II. Design

The design of hardware-based methodologies for forensics analysis or malware detection involves three parts: (1) designing the system architecture of the approaches, which determines how data should be collected, (2) selecting objects of interest according to the specific objective, and (3) selecting appropriate analysis methods to process the collected data for the corresponding purpose.

A. System Architecture

Generally speaking, a complete system of a hardware-based approach consists of two main components, namely a data logging component and a data analysis component. The data logging component collects data of interest exclusively from hardware and has to be implemented in hardware, integrated with the underlying microprocessor. Unlike software-based approaches, in this way, there exists no physical pathway for the OS, hypervisor, or any application running on the system to interfere with the logged data, ensuring resistance to software tampering.

The data analysis component, on the other hand, is responsible for performing specific analyses on the logged data, while its actual implementation depends upon the type of defense mechanism to be employed. As stated in Section I, a defense mechanism branches into either a retroactive investigation or a real-time detection. In the former case, since the retroactive investigation applies *ex post facto* analysis, the logged data can be exported through a secure hardware channel to a trusted software environment, where the data analysis component can be implemented *off-line* with flexibility, e.g. choice of analysis methods, adaptive analysis models, etc. On the other hand, real-time detection requires prompt *on-line* reaction when malicious events occur. As a result, the data analysis component is generally also implemented in hardware, deeply integrated with the data logging component, to ensure both security and a timely response. Essentially, on-line methods detect and/or prevent malicious events at an early stage, at the cost of loss of flexibility and increased design overhead. Figure 1 illustrates the general system architecture that a hardware-based forensics or malware detection method may follow.

B. Object of Interest

Depending on their core objective, state-of-the-art forensics or malware detection methods can be categorized into *data-centric* or *program-centric*. Data-centric methods generally focus on the integrity of a piece of specific data in order to investigate whether any unauthorized modification occurs and disable it. Given the nature of malicious software, popular objects of interest of methods in this category include kernel image, kernel service table, control flow of kernel service, network channel, etc. Program-centric methods, on the other hand, model the expected behavior of a program in order to identify what program it actually is or whether it is malicious. The OS-level abstraction of a program, i.e. *process*, is the common object of interest of methods in this category.

C. Analysis Method

Similar to the object of interest, existing forensics or malware detection methods employ various analysis methods, which fall into either *signature-based* methods or *behavior-based* methods. Signature-based methods generate *checksums* over their objects of interest, which can be used as a golden reference for integrity checking or as a description of the expected behaviors. These methods benefit from their simplicity of implementation and may work well with those objects whose execution is fixed or infrequently changed. Given the complexity of program execution, however, behavior-based methods are more favorable when a process is the object of interest. These methods aim at modeling program behavior dynamically based on a number of features. In order to allow enough flexibility to account for program execution variation and, at the same time, be able to distinguish benign from malicious program behavior, machine learning and/or statistical analysis is typically employed.

Overall, a hardware-based approach for forensics analysis/malware detection can be constructed through combination of these three dimensions, i.e. off-line/on-line, data-centric/program-centric, signature-based/behavior-based.

III. Case Study

In this section, we discuss two incarnations of the proposed concept to illustrate the effectiveness of hardware-based methodologies. Specifically, we present an off-line, program-centric, behavior-based workload forensics method, as well as an on-line, data-centric, signature-based intrusion detection method.

A. Workload Forensics

1) Implementation: The primary objective of the workload forensics method proposed herein is to develop a system-level workload reconstruction capability, which reconstructs workload at the granularity of a process [9]. This method, therefore, follows an off-line system architecture, which implements the data logging in hardware while analyzing collected data in software. When developing a hardware-based data logging solution, the semantic gap problem needs to be addressed. Indeed, we need to identify a process directly at the circuit level (i.e. without relying on data available at the OS level), so that we can associate with it the logged information that will be used for workload reconstruction. Fortunately, thanks to the work in [13], the CR3 register of x86 resolves this problem, as changes in the CR3 value perfectly match the events of process creation, switching and termination. Accordingly, by monitoring the CR3 register, delineating processes becomes possible, thereby bridging the semantic gap.

To describe the behavior of the process, this approach uses user-space instructions causing iTLB misses as the logging object, inspired by the fact that program execution has phases [14]. The reason of selecting TLB profile is as follows. A TLB is a small cache memory which maintains recent translations of virtual addresses to physical addresses. The actual TLB implementation is split into two parts, one for instruction addresses (iTLB) and the other for data access addresses (dTLB). In x86, when the CR3 value changes, the entire TLB is flushed. This design convention benefits the approach in two ways. First, all TLB events can be accurately associated with the process represented by the current CR3 value. Second, the effect of different order of program execution is mitigated, as the TLB starts fresh with every process. Therefore, the granularity of the logged data (i.e., process-level) matches our analysis target.

The actual behavior model employs multiple descriptive features generated from the raw TLB profile, as shown in Figure 2. Specifically, this approach initially splits the raw TLB profile into partitions, where counts of occurrences of 6 types of operators as well as 12 types of operands are extracted as features. The operator (Op.) types include:

1) **Data Op.**: operations performing data manipulation, such as storing/loading values, setting flags, etc.
2) **Stack Op.**: operations performing stack manipulation.
3) **ALU Op.**: operations performing arithmetic or logic calculation.
4) **Control Flow Op.**: operations changing instruction execution flow.
5) **I/O Op.**: operations working with x86 I/O ports and interacting with peripherals.
6) **Floating Point Op.**: operations performing all FP related manipulation.

The remaining 12 operand (Opr.) features include 8 features corresponding to the 8 general purpose registers of 32-bit x86, one for memory reference, one for XMM registers and floating point stack, one for all segment registers, and one for immediate value. Ultimately, a vector $F.V._i = < Op._1, ..., Op._6, Opr._1, ..., Opr._{12} >$ is generated for each partition. For each process, as identified through its CR3 value, a list of feature vectors $[F.V._1, ..., F.V._i, ...F.V._{end}]$ is collected, reflecting the order of partitions.

As introduced in Section II-C, behavior-based methods generally apply machine learning for analysis. Given the objective of workload reconstruction, this method employs multi-class classification, where each class corresponds to a single process. Additionally, previously unseen processes are identified through outlier detection. Regarding process

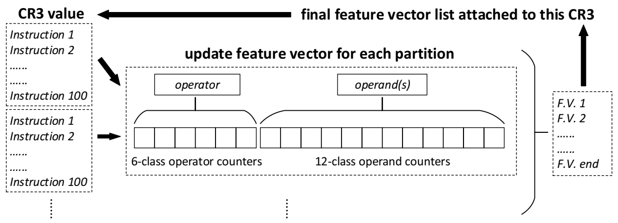


Fig. 2: Feature extraction mechanism

classification, we experimented with two different non-linear multi-class classifiers of varying complexity and performance, namely K-Nearest Neighbors (KNN) and Support Vector Machine (SVM). To perform outlier screening, we leverage the probability estimation available in the SVM. Given a sample, the SVM provides not only the chosen class, but also a vector containing the probabilities that this sample belongs to each known class. The *conjecture* of the outlier detection method is that when the sample comes from a known distribution (i.e., previously seen), the probability of the winning class will dominate all others, while when it comes from an unknown distribution (i.e., outlier), multiple classes will exhibit fairly similar probability. Therefore, a simple outlier screening criterion is the probability difference between the first and second most likely classes. If this difference exceeds a threshold, which can be learned through cross-validation, the process is classified as an outlier. In our implementation, we used KNN from the Matlab library and SVM from the LIBSVM library [15].

2) Evaluation: Evaluation of this workload forensics method was performed in Simics, wherein we simulated a 32-bit x86 machine with a single Intel Pentium 4 core running at 2Ghz and containing 4GB of RAM, on which we loaded a minimum installation Ubuntu server that embeds a Linux 2.6 kernel as the OS. All collected data is normalized and fed to the analysis software via Python/Matlab. We use MiBench [16], a free commercially representative benchmark suite as our workload, which contains tens of application classes. The entire suite was executed 100 times, with each application invoked with various valid arguments or in the background (& option). The workload execution was also randomized to avoid the bias that a specific order might impose.

The process classification results using KNN and SVM are shown in Table I. As may be observed, both classifiers performed very well in correctly classifying the processes, reaching an overall classification accuracy of 96.97% and 96.63% respectively. For most classes, this accuracy was even higher. However, parasite processes such as `savelog`, `cmp`, etc., can be created sporadically during the execution of MiBench applications in our simulation environment. Samples of these processes were considered in our experiments but their low frequency of occurrence limits the available samples and undermines the corresponding classification accuracy. Fortunately, considering their weight, their overall impact on process classification accuracy is small.

To evaluate the effectiveness of our system in identifying previously unseen processes, we repeated the experiment,

application class	training samples	testing samples	KNN accuracy	SVM accuracy
overall	2386	2376	96.97%	96.63%
bash	1088	1087	100%	100%
cjpeg	25	25	100%	100%
djpeg	25	25	96%	100%
search	50	50	98%	98%
tiff2rgba	50	50	100%	100%
tiffmedian	50	50	96%	100%
toast	50	50	96%	96%
		......		
dpkg	11	11	72.73%	72.73%
savelog	9	9	55.56%	55.56%
cron	4	3	66.67%	66.67%
cmp	3	3	33.33%	33.33%

TABLE I: Process classification accuracy (subset of classes)

with 5 randomly selected classes omitted from the training set, while retaining them in the testing set to mimic outlier processes. Through cross-validation, we set the threshold for outlier screening to 0.6, which is applied to the processes in the testing set. Ultimately, we observed that even the simple outlier screening method described above can achieve high outlier detection accuracy, with the average false positive (i.e. seen process classified as outlier) and false negative (i.e. outlier classified as seen process) rate at 12.31% and 5.13%, respectively. Nevertheless, advanced outlier detection methods can further improve the results.

Since the only hardware component involved in this method is the logging component, we evaluate the overhead of this method in term of required data logging rate, which is calculated as follows. For each partition of a process, our method requires one feature vector containing 18 elements. If we assume `partition_size` to be 100, as in our experiments, we only need 7 bits for each element. The number of partitions generated per second is determined by the iTLB miss rate. Assuming clock cycles per instruction (CPI) has an optimal value of 1, the estimated logging rate is calculated step by step by the equations below:

$$F.V.\ size = 18 \times \lceil \log_2 partition\_size \rceil \quad (1)$$

$$partition\ generation\ rate = \frac{iTLB\ miss\ rate}{partition\_size} \quad (2)$$

$$bits/inst. = F.V.\ size \times partition\ generation\ rate \quad (3)$$

$$est.\ logging\ rate(bits/sec) = \frac{bits/inst. \times clk\ freq.}{CPI(assumed = 1)} \quad (4)$$

We ran our benchmark suite several times to obtain an average iTLB miss rate, the value of which was 0.0016%, resulting in an estimated data logging rate of only 5.17 KB/sec. While a typical TLB miss rate is expected to be around 0.01-1% [17], since we consider only user-space virtual addresses and only iTLB misses, the relevant miss rate for our scheme is much less. Furthermore, since we assumed an optimal CPI of 1, the logging rate should be even lower in realistic cases.

B. Intrusion Detection

1) Implementation: This method proposes a hardware-based on-line intrusion detection to ensure integrity of an executed system call service routine in a way that is immune to tampering by software, hence, follows the on-line system architecture. The idea is motivated by the fact that most malware detection methods rely on system call-related information, yet their logging/monitoring mechanisms rarely inspect the actual system call execution flow, which leaves room for malwares to evade detection through *system call hijacking*.

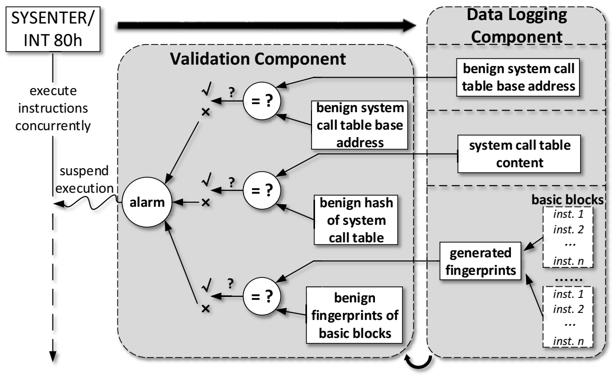
System call hijacking enables an attacker to control the execution flow of one or several system calls; thereby, malicious code can be introduced and executed without the knowledge of the legitimate system user. In Linux OS, for example, this can be achieved through a kernel rootkit exploiting the Loadable Kernel Module (LKM), which is intended to extend or customize the functionality of the original kernel. When the extended or customized functionality is no longer required, the kernel module can be unloaded, restoring the kernel to its original state and, thereby, leaving no trace. This work focuses on three types of system call hijacking:

1) **System Call Table Redirection:** The type 1 attack redirects the OS to a different system call table controlled by the attacker when a system call is invoked.
2) **System Call Table Modification:** The type 2 attack modifies the value of certain entries in the original system call table so that the attacker can redirect the service routine of certain system calls to his/her own malicious code snippet.
3) **Service Routine Modification:** The type 3 attack directly modifies the service routines of one or more system call.

Theoretically, the type 1 attack requires the least design complexity but suffers more risk to be detected, while the type 3 attack is the most complicated by design yet the most likely to evade detection.

The proposed hardware-based intrusion detection system, which consists of two components both implemented in hardware, i.e., a data logging component and a validation component, addresses the three types of system call hijacking separately. In particular, the data logging component collects three critical pieces of information related to the integrity of system call execution, namely the base address of the system call table, the contents of the system call table, and the actual system call service routines. When a system call is invoked, the validation component, then, contrasts this information against their corresponding valid signatures, stored in the hardware as well, in order to detect the three types of system call hijacking attacks *in real time*.

Considering the design cost, it is not feasible to store the uncompressed benign system call table content and corresponding service routines as valid signatures in hardware. Instead, this method employs a Multiple Input Signature



SYSENTER/ INT 80h
execute instructions concurrently
suspend execution
Data Logging Component
Validation Component
benign system call table base address
system call table content
basic blocks
inst. 1
inst. 2
inst. n
generated fingerprints
benign system call table base address
benign hash of system call table
benign fingerprints of basic blocks
alarm
= ?

Fig. 3: High-level design of the hardware-based on-line intrusion detection

Register (MISR) to compress and generate fingerprints of this critical information. Using a MISR for our purpose has three advantages: (1) the hardware structure of a MISR is relatively simple, involving only D-Flip-Flops (DFF) and XOR gates, whose interconnection is defined by a characteristic polynomial; thus, it incurs low design overhead. (2) A MISR is scalable and can process multiple inputs simultaneously, independent of the input length; this allows us to efficiently process entire table contents or instruction sequence in order to generate fingerprints. (3) The MISR has a relatively low aliasing probability. Aliasing occurs when identical signatures are generated for different input sequences and can undermine our ability to detect invalid instruction sequences. An approximation of the aliasing probability of a MISR is 2^{-n}, where n is the degree of its characteristic polynomial.

Generating fingerprints of system call table content using a MISR is relatively straightforward while it is slightly more involved in the case of system call service routines. Specifically, our method generates a fingerprint for each basic block of the executed routine, which is compared afterwards against a set of known acceptable fingerprints for basic blocks of this routine. The choice of a basic block, as opposed to the entire system call service routine, as the minimum entity to be fingerprinted is driven by practicality. A basic block is a snippet of atomic code executed between two control flow transfers. Therefore, the actual instructions executed are fixed, hence the golden fingerprint of a basic block can be statically computed. In contrast, the instructions executed by the entire system call service routine depend on the arguments with which it is invoked at run-time; hence there is a multitude of golden fingerprints which are not only harder to exhaustively identify but may also leave more room for malicious modifications to go undetected. A high-level architecture of the proposed method is depicted in Figure 3.

As stated above, fingerprinting the system call service routine produces sets of multiple golden fingerprints, therefore, requiring a space-efficient storage solution. To this end, this method employs a Bloom filter for compactly storing the golden fingerprints and rapidly performing membership tests. A Bloom filter is a space-efficient probabilistic data structure, used to test whether an element is a member of a set [18]. A typical Bloom filter consists of an m-bit array and implements k different hash functions h_i, $i \in [1, k]$, each of which maps an input element E to one of the positions in the array through a uniform random distribution. An empty Bloom filter is an array with all 0s. When adding an element, all k position bits mapped by the hash functions for this element are set to 1. As a result, an element is a member of the set if and only if $\forall i \in [1, k], h_i(E) = 1$. Accordingly, a fingerprint of a basic block of the actual system call routine is valid if and only if its corresponding position bits remains all 1s. Figure 4 illustrates the hardware implementation of the validation process.

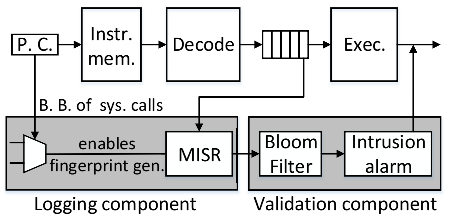


Fig. 4: Architecture of proposed method

TABLE II: Kernel rootkit detection summary

rootkit	description	detected?
Type-1 attack	system call table redirected	✓
Type-2 attack	table entry sys_open() redirected	✓
Type-2 attack	table entry sys_write() redirected	✓
Type-2 attack	table entry sys_mkdir() redirected	✓
Type-3 attack	sys_write() routine modified	✓

2) Evaluation: Evaluation of this hardware-based intrusion detection method was also performed in Simics with the same configuration as the one in the case of workload forensics. To evaluate effectiveness of this method, we first perform static analysis to collect the golden fingerprints of the basic blocks for each system call service routine, and to program the corresponding Bloom filters. Next, we use MiBench to collect fingerprints for legitimate workload. On the other hand, to collect fingerprints for contaminated workload, five rootkits which exploit LKM to launch system call hijacking attacks of the three types introduced in Section III-B1 were implemented based on the Linux rootkit template maKit[2].

Table II summarizes the five kernel rootkits which we used to launch system call hijacking attacks. Following the definitions in Section III-B1, the first one is Type 1, the next three are Type 2 and the last one is a Type 3 threat. All five were successfully detected by this method, as they invoked system calls whose service routine execution generated fingerprints that were rejected by the corresponding Bloom filter.

The incurred hardware overhead of this method, which is evaluated using a predictive 45nm Process Design Kit (PDK) [19], is dominated by the implementation of MISR and the hash function in Bloom filter. Compared with a 45nm Intel Processor[3], whose area is 107 mm^2 and average power consumption is 65 W, the total overhead of this hardware implementation is negligible, as summarized in Table III.

IV. Related Work

In this section, we briefly present several other efforts in the hardware-based forensics and/or malware detection. For

[2]https://github.com/maK-/Syscall-table-hijack-LKM
[3]http://ark.intel.com/products/35605

TABLE III: Design overhead summary

	area (μm^2)	**power** (mW)
logging	1810.56	5.55
validation	7419.16	15.9
total	**9229.72**	**21.45**
microprocessor	107×10^6	65×10^3
overhead	**0.008625%**	**0.033%**

example, similar software-based data-centric signature-based methodologies, such as Control Flow Integrity (CFI) checking, can be applied at the hardware level, which are expected to incur less runtime overhead [11], [20], [21]. Furthermore, in contrast to traditional CFI methods, CFIMon uses performance counters to model code execution behavior and detect control flow deviation [22], which falls into the data-centric behavior-based methodologies. Nevertheless, these methods generally require specific support from the underlying OS or compiler to bridge the semantic gap [20], [21].

While system call-related information can also benefit hardware-based program-centric approaches in malware detection [23], most of the approaches in this category try to leverage low-level information extracted directly through hardware in order to model the program behavior and perform forensic analysis and/or malware detection. For instance, performance counters can be used to model program behavior through machine learning methods, based on which on-line malware detection can be performed [8], [24]. Besides performance counters, an alternative method may also collect low-level architectural information, e.g. memory address reference, instruction opcode, etc., to model program behavior and perform malware detection [10].

V. Conclusion

We explored the possibility of performing workload forensics and/or malware detection through hardware-based logging and/or analysis mechanisms. Unlike traditional software-based methods, a hardware-based approach benefits itself from its innate immunity to software tampering, which ensures the security and reliability of the logging and analysis system. We introduced a general architecture which the hardware-based forensics/malware detection systems need to follow, whose possible implementations can be constructed through combinations of three dimensions (i.e. on-line/off-line, data-centric/program-centric, signature-based/behavior-based), depending on their various purposes. We illustrated this general concept through two incarnations, the first of which performs a hardware-based workload reconstruction through TLB profiling while the second of which performs a hardware-based on-line intrusion detection through system call fingerprinting. Experimental results corroborate that even low-cost hardware implementations can facilitate highly successful forensics analysis and/or malware detection.

References

[1] L. Garber, "Encase: A case study in computer-forensic technology," *IEEE Computer Magazine*, Jan. 2011.

[2] J. Criswell, N. Dautenhahn, and V. Adve, "Kcofi: Complete control-flow integrity for commodity operating system kernels," in *Proc. of the 2014 IEEE Symp. on S & P*, 2014, pp. 292–307.

[3] T. Bletsch, X. Jiang, and V. Freeh, "Mitigating code-reuse attacks with control-flow locking," in *Proc. of the 27th Annual Computer Security Applications Conf.*, 2011, pp. 353–362.

[4] 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*, vol. 7, no. 6, pp. 1876–1889, 2012.

[5] 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*, vol. 35, 2016, pp. 485–498.

[6] 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*, 2012, pp. 586–600.

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

[8] J. Demme, M. Maycock, J. Schmitz, A. Tang, A. Waksman, S. Sethumadhavan, and S. Stolfo, "On the feasibility of online malware detection with performance counters," in *40th Annual Intl. Symp. on Computer Architecture*, 2013, pp. 559–570.

[9] L. Zhou and Y. Makris, "Hardware-based workload forensics: Process reconstruction via TLB monitoring," in *IEEE Intl. Symp. on Hardware Oriented Security and Trust*, 2016, pp. 167–172.

[10] M. Ozsoy, K. N. Khasawneh, C. Donovick, I. Gorelik, N. Abu-Ghazaleh, and D. V. Ponomarev, "Hardware-based malware detection using low level architectural features," in *IEEE Trans. on Computers*, vol. PP, no. 99, 2016.

[11] A. Kanuparthi, J. Rajendran, and R. Karri, "Controlling your control flow graph," in *IEEE Symp. on Hardware Oriented Security and Trust*, 2016.

[12] C. Song, B. Lee, K. Lu, W. Harris, T. Kim, and W. Lee, "Enforcing kernel security invariants with data flow integrity," in *The Network and Distributed System Security Symp.(NDSS)*, 2016.

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

[14] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder, "Discovering and exploiting program phases," *IEEE Micro*, vol. 23, no. 6, pp. 84–93, 2003.

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

[16] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *IEEE Intl. Workshop on Workload Characterization*, 2001, pp. 3–14.

[17] D. A. Patterson and J. L. Hennessy, *Computer Organization And Design. Hardware/Software Interface. 4th edition*. Morgan Kaufmann, 2009.

[18] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[19] 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*, 2007, pp. 173–174.

[20] L. Davi, P. Koeberl, and A. Sadeghi, "Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation," in *Proc. of the 51st Annual Design Automation Conference*, 2014, pp. 1–6.

[21] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Transparent rop exploit mitigation using indirect branch tracing," in *22nd USENIX Security Symp.*, 2013, pp. 447–462.

[22] Y. Xia, Y. Liu, H. Chen, and B. Zang, "Cfimon: Detecting violation of control flow integrity using performance counters," in *Proc. of the 2012 42nd Annual IEEE/IFIP Intl. Conf. on Dependable Systems and Networks*, 2012, pp. 1–12.

[23] S. Das, Y. Liu, W. Zhang, and M. Chandramohan, "Semantics-based online malware detection: Towards efficient real-time protection against malware," *IEEE Trans. on Information Forensics and Security*, vol. 11, no. 2, pp. 289–302, 2016.

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