Hardware-based Workload Forensics: Process Reconstruction via TLB Monitoring

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

***Abstract*—We introduce a hardware-based methodology for performing workload execution forensics in microprocessors. More specifically, we discuss the on-chip instrumentation required for capturing the operational profile of the Translation Lookaside Buffer (TLB), as well as an off-line machine learning approach which uses this information to identify the executed processes and reconstruct the workload. Unlike workload forensics methods implemented at the operating system (OS) and/or hypervisor level, whose data logging and monitoring mechanisms may be compromised through software attacks, this approach is implemented directly in hardware and is, therefore, immune to such attacks. The proposed method is demonstrated on an experimentation platform which consists of a 32-bit x86 architecture running Linux operating system, implemented in the Simics simulation environment. Experimental results using the Mibench workload benchmark suite reveal an overall workload identification accuracy of 96.97% at an estimated logging rate of only 5.17 KB/sec.**

I. Introduction

As reliance of our everyday lives on technology continues to increase, so does the amount of sensitive information that is stored, processed and communicated in electronic form. Inevitably, this also attracts intensified efforts to gain unauthorized access to such information for monetary, political, or other benefit. As a result, when such malicious acts occur, the ability to retroactively investigate and identify the events that led to the compromising of sensitive data becomes invaluable.

Workload forensics generally describes the process of collecting and analyzing data related to past execution of computer programs, in order to understand and/or reconstruct the events that transpired. Such forensic analysis methods are broadly categorized into data-centric and program-centric. The former typically employ state snapshots to track changes that occur to objects of interest, such as files, in order to enable data recovery and intrusion detection [1], [2], [3], [4], [5], [6]. The latter rely on extracted signatures or behavioral models to distinguish between programs, in order to enable malware detection or workload reconstruction [7], [8].

Numerous methods from both categories have been developed at the operating system (OS) level, leveraging its rich semantic information and flexibility [9], [10], [11]. OS-level methods, however, can be subject to software attacks staged at the same level. Kernel rootkits, for example, may be used to compromise an OS-level logging system and eliminate all traces associated with malicious actions. In order to overcome this limitation, hypervisor-level forensics solutions have been proposed. A hypervisor, a.k.a. virtual machine monitor (VMM), is a software which 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 solutions. Therefore, data collected by a forensics method at the hypervisor level is generally immune to OS-level software attacks. Nevertheless, as shown through recent work [12], the hypervisor itself can be the target, as several vulnerabilities and intrusion methods have been identified. As a result, similar attacks compromising integrity of the logged forensics data to conceal malicious events can be staged at the hypervisor level.

In contrast, in this work we explore the possibility of relying exclusively on data collected directly through the hardware, without the intervention of a hypervisor or an OS whereby the logged information may be compromised. In essence, we seek to leverage the fact that – to our knowledge – it is not possible to hide from the hardware the actions of any executed software, even software that seeks to hide itself. Accordingly, traces obtained from hardware are expected to be immune to software-based tampering. At the same time, however, a hardware-based forensics solution requires circuitry addition and modification for identifying, extracting, and logging the relevant information. Therefore, judicious selection of minimal information sufficient for fulfilling the targeted task becomes critical. Herein, we investigate the feasibility and effectiveness of a hardware-based system which seeks to reconstruct executed workload at the granularity of a single process, through minimal information obtained from the Translation Lookaside Buffer (TLB). This idea is demonstrated through execution of the Mibench benchmark suite on an x86 architecture running Linux OS, implemented in the Simics simulation environment.

The remainder of the paper is structured as follows. In Section II, we discuss related work. The proposed method is introduced in Section III, while details of the implementation are provided in Section IV. Experimental results evaluating the accuracy of the proposed method in reconstructing workload, as well as its overhead are presented in Section V and conclusions are drawn in Section VI.

II. Related Work

The state-of-the-art in forensic analysis methods found in the literature can be broadly categorized, based on the level at which they are implemented, into OS-level approaches and hypervisor-level approaches. Within each category, existing methods can be further divided into data-centric and program-centric, depending on the core object of the forensic analysis. Table I provides a taxonomy of all related methods described in this Section, including the method proposed in this paper.

A. *OS level approaches*

OS-level approaches generally benefit from semantic-rich information, such as process ID, system call sequence, etc., which is available at this level. Data-centric approaches in this category mainly focus on the integrity of file system objects, such as files on disks. Various commercial products fall into this paradigm. For example, EnCase [1] creates images for disk data to enable data recovery and/or to ensure data integrity. Similar products include FTK [2] and Registry Recon [3]. Program-centric approaches, on the other hand, seek to model program behavior based on a number of different features. For example, the system call number and its corresponding argument have been widely used as such 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.

A large body of work on intrusion detection follows this paradigm [9], [10], [11]. In general, these methods rely solely on analysis of system call sequences. An interesting extension is introduced in [13], which focuses on a subset of system calls that are deemed to be most informative. Clustering of system call arguments is also employed in order to better understand how it has been invoked by the operating system. In another incarnation, called Accessminer [14], further information such as timestamps, return values, etc., is used to model how benign programs access OS resources (e.g. files and registry entries), so that malware-induced suspicious behavior can be better distinguished from normal functionality.

B. *Hypervisor-level approaches*

Hypervisor-level approaches benefit from the inherently higher security offered by the virtualization and the isolation that the hypervisor provides, as we discussed in Section I. As a trade-off, however, approaches at this level now suffer from the semantic gap problem. Specifically, while methodologies similar to those introduced at the OS-level can be applied at the hypervisor-level, we first need to interpret the information collected at the hypervisor level and bridge the semantic gap by linking this information to tangible OS-level objects. To achieve this, architecture-specific hardware conventions are typically relied upon. For instance, Antfarm [15] uses the CR3 register available in the x86 architecture in order to identify process creation, switching and termination. By convention, the CR3 register stores the base address of the page table directory of the currently active process. This binding provides a view of all process handling events. Most hypervisor-level approaches rely on the CR3 value in order to understand the life-cycle of a fundamental OS-level object, namely a process.

Once the semantic gap is bridged, program-centric methods similar to the ones developed at the OS-level may be applied. For example, the system call number/sequence can be extracted from the instruction flow and specific registers (rather than from a software tracing tool, such as `strace`), in order to perform behavior-based modeling and analysis [7], [8]. Data-centric methods may also be devised. Methods along this direction monitor the critical area in kernel memory (e.g. system call table, kernel text, etc.) in order to prevent malicious changes therein [4]. Such methods even go to a lower layer, to check whether contents on the disk and its image in main memory match [5], [6]. Nevertheless, they still rely on OS-level information (e.g. `system.map`) to locate which part is critical to keep their eyes on [6].

	Data-centric	**Program-centric**
OS-level	[1],[2],[3]	[9],[10],[11],[13],[14],[16]
Hypervisor-level	[4],[5],[6]	[7],[8],[15]
Hardware-level	N/A	[17], **this work**

TABLE I: Taxonomy of related work and proposed method

Besides using system call related information to model program behavior, the idea of phase-based behavior modeling has also been investigated. The underlying conjecture is that program execution exhibits repeating patterns (phases), which can be used to model and predict its behavior [16]. A recent approach also investigates the use of performance counters to model program behavior for malware detection purposes [17].

III. Proposed Method

The primary objective of the method proposed herein is to develop a system-level workload reconstruction capability, which can be used for the purpose of forensics. In contrast to OS-level and hypervisor-level approaches, however, this method should be immune to tampering by software. To this end, we introduce a hardware-based solution, wherein the information required for reconstructing the workload is obtained and stored directly in the hardware. 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. To extract the logged data from the processor, our approach also relies on a dedicated port, which is invisible to and inaccessible by the OS. Using this port, the data can be continuously off-loaded to a secure storage or directly fed into a trusted environment where the forensics analysis will be performed using statistical methods. A top-level view of the proposed system architecture is shown in Figure 1.

In this work, we experiment with one simple instantiation of this general idea. Specifically, we explore the possibility of reconstructing workload at the granularity of a process, while relying solely on information available through monitoring the TLB and statistically processing this information. Considering our objective of developing a hardware-based solution, however, we need to address the semantic gap problem. 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 [15], 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.

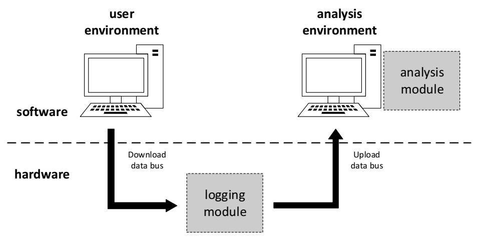


Fig. 1: High-level system architecture

Below, we provide details of the two key components of our system, namely the logging module and the analysis module.

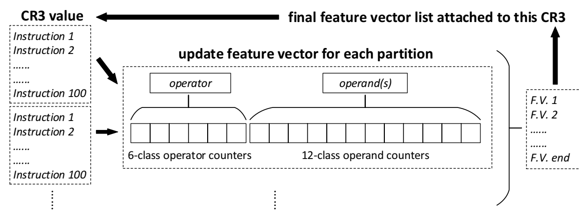
A. Logging module

Program execution typically follows phases, which can be effectively predicted via performance counter values [17]. Performance counters, however, generally contain global values, reflecting performance of a microprocessor over its entire workload. Moreover, order of program execution will affect performance counter values. As a result, bridging the semantic gap and associating these values accurately with OS-level objects, such as processes, is not at all straightforward.

1) Logging object: To address this limitation, rather than using performance counter values, our approach uses instructions causing TLB misses as its main logging object. A TLB is a small cache memory which maintains recent translations of virtual addresses to physical addresses. In x86, when the CR3 value changes, the entire TLB is flushed. This design convention benefits our 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.

In x86, the TLB is split into two parts, one for instruction addresses (iTLB) and the other for data access addresses (dTLB). The logging module monitors the iTLB state and identifies the instructions which raise an iTLB miss. Only user-space instructions are considered in our scheme. In the Linux OS, all virtual addresses higher than `0xC0000000` are regarded as pointers to kernel space. Accordingly, our logging module ignores iTLB miss events raised by such addresses. In the end, each CR3 value, which represents a separate process, can be associated with a sequence of instructions (which caused iTLB misses). Figure 3 shows the logging logic.

2) Feature extraction: In order to use machine learning for analysis, we extract a normalized set of features from the logged data. In our scheme, we use features which reflect both order and frequency information. Conceptually, for each CR3 value, its associated set of instructions causing iTLB misses is first partitioned into subsets of a maximum size of `partition_size`. Partitioning helps retain order information while reducing log size. In one extreme, choosing `partition_size` to be 1 retains all instruction order information but is too expensive and, most likely, unnecessary. In the other extreme, no partitioning would minimize the log size but would also sacrifice all order information, thereby limiting the accuracy of the forensic analysis. In our system, we experimented with `partition_size` of 100 instructions. In practice, to minimize the required hardware, we do not log the actual instructions in each partition but, rather, a set of 18 frequency features. These 18 features are extracted through counters which are updated every time a qualifying iTLB miss occurs, and reflect information regarding the *operator* and the *operands* of the qualifying instruction, as shown in Figure 2.


CR3 value
final feature vector list attached to this CR3
Instruction 1
Instruction 2
......
......
Instruction 100
update feature vector for each partition
operator
operand(s)
6-class operator counters
12-class operand counters
F.V. 1
F.V. 2
......
......
F.V. end
Fig. 2: Feature extraction mechanism

The first six features capture the count of qualifying instructions for each of the following operator (Op.) types:

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 twelve features capture the count of qualifying instructions which use the various types of operands (Opr.). These 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.

A vector $F.V._i =< Op._1, ..., Op._6, Opr._1, ..., Opr._{12} >$ is extracted 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. The length of this list is considered as an additional feature. Ultimately, a feature matrix is generated, as shown in Figure 2. We note that, since the number of partitions can vary from process to process, once the data is off-loaded to the analysis module and prior to statistical processing we use zero padding for the feature lists of processes so that all lists have the same number of columns in the feature matrix.

B. Analysis module

The objective of the analysis module is to reconstruct workload execution at the granularity of a process, using the extracted feature matrices. Since forensics is typically an *ex post facto* effort, analysis is implemented in software and is executed in a trusted environment. However, future extensions could use dedicated on-chip learning to perform the analysis directly in hardware, possibly even in real-time, in a fashion similar to the malware detection method described in [17].

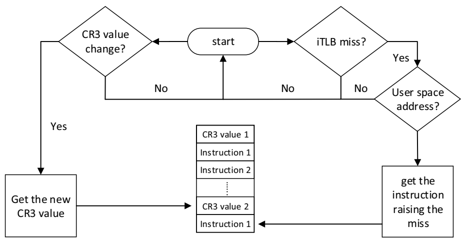


Fig. 3: Logging logic

The actual analysis is based on machine learning and employs multi-class classification, where each class corresponds to a single process. Additionally, previously unseen processes are identified through outlier detection. 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). KNN computes the k nearest neighbors for a sample based on their Euclidean distance and assigns the sample to a class based on majority voting among these neighbors. SVM, on the other hand, generates decision boundaries which separate the feature space into labeled sub-spaces, while ensuring maximal separation among them. When evaluating a new sample, the SVM classifies it based on the label of the sub-space that it falls into. An important consideration when applying machine learning is the high dimensionality of the feature matrix. Since the extracted feature vector list may contain a large number of elements, it is necessary to reduce the dimensionality before performing classification, in order to avoid the curse of dimensionality. To this end, we use Principal Component Analysis (PCA), which generates a lower-dimensional feature matrix, while retaining most of the information of the original matrix. In our implementation, we used KNN from the Matlab library and SVM from the LIBSVM library [18].

IV. Logging System - Hardware Implementation

As mentioned earlier, our logging mechanism resides entirely in hardware, therefore requiring modification in CPU design, in order to eliminate the possibility of software attacks. To minimize the required storage for the data log, feature extraction is also implemented in hardware, with the final log containing only the feature matrices.

The hardware logging module consists of three main components, with its overall architecture shown in Figure 4:
Event Monitor: this component is used to monitor critical events, including TLB miss, CR3 register update, program counter update, etc. The event monitor serves as the main controller of the entire logging system. In x86, the TLB is implemented in the Memory Management Unit (MMU) and miss events are handled transparently by the hardware. The event monitor is expected to reside in the CPU but is also connected to the iTLB cache memory to get notification when a miss occurs. After the hardware resolves this miss (and independently of whether a translation is found in the page table or not), the event monitor picks up the instruction which raised the iTLB miss and feeds it to the feature generator. In parallel, the value of the CR3 register, which works as an

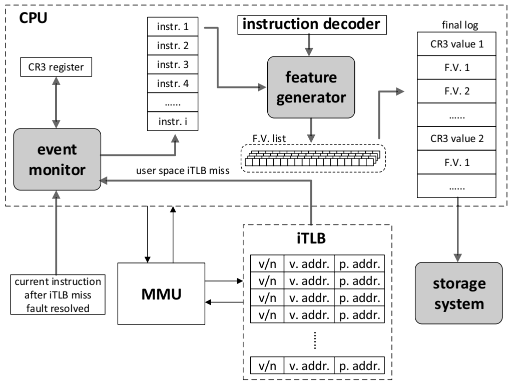


Fig. 4: Logging system implementation in hardware

identifier of the current process, is monitored to ensure that the current iTLB miss event is associated with the correct process.
Feature Generator: this component performs feature extraction for each instruction which raises an iTLB miss. During decoding of such an instruction, the feature generator produces the corresponding feature list according to the rules introduced in Section III-A2. A temporary register is used to update the values of a feature vector. When the partition size limit is reached or the current process terminates, the final value is sent to the storage system along with the CR3 value.
Storage System: this component is the actual space where the logged information is stored. A FIFO buffer is used to handle the clock difference between the CPU and the storage system. To save memory space, zero padding is not done in hardware. Instead, the size discrepancy between log entries is handled during analysis. Periodically or continuously the logged data is transmitted through a dedicated port, which is physically inaccessible by the OS, to a trusted external storage or to the environment where analysis is performed.

V. Experimental Results

We now proceed to assess the effectiveness of our method in correctly classifying known processes and identifying previously unseen ones. Additionally, we evaluate the data logging rate required, as this reflects the incurred hardware overhead.

Our experiments were 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 our operating system. All collected data is normalized and fed to the analysis software via Python/Matlab.

A. Process classification accuracy

To evaluate the accuracy of our method in correctly classifying processes, we use MiBench [19], a free commercially representative benchmark suite as our workload, which contains a few tens of application classes. The entire suite was executed 100 times, with each application invoked with various valid arguments or in the background (& option). We also randomized workload execution to avoid the bias that a specific order might impose. We exploit the Simics feature, *haps*, to hook our event monitor on the iTLB and the program counter. Our feature extraction method was then applied on

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%
susan	75	75	100%	100%
search	50	50	98%	98%
madplay	50	50	96%	96%
tiff2bw	50	50	98%	94%
tiff2rgba	50	50	100%	100%
tiffmedian	50	50	96%	100%
basicmath	50	50	92%	90%
toast	50	50	96%	96%
untoast	50	50	94%	94%
rawcaudio	25	25	92%	92%
rawdaudio	25	25	52%	52%
		......		
run-parts	18	18	83.33%	83.33%
date	15	15	86.67%	86.67%
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%

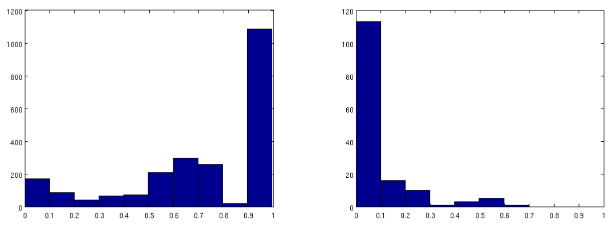
TABLE II: Process classification accuracy (subset of classes)

the workload log. In total, we collected a dataset containing 4762 samples, each comprising a feature vector matrix and representing a process to be classified. Initial dimensionality of the feature vector matrix was as large as 83612 and was reduced to 200 after applying PCA. The reduced matrix was then fed into the two classifiers. Half of the samples of each application class were used for training and the other half for testing. The process classification results using KNN and SVM are shown in Table II. 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`, `cron`, and `cmp`, 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.

A noteworthy exception is the process `rawdaudio`, for which half of the instances are misclassified as `rawcaudio`, despite the adequate number of training/validation samples. This is explained by the fact that `rawcaudio` implements an Adaptive Differential Pulse Code Modulation (ADPCM) encoding algorithm, wherein `rawdaudio`, which implements the corresponding decoding algorithm, is invoked as a major functional unit. This inclusion introduces similarity and reduces classification accuracy for `rawdaudio`. Additional features of more advanced machine learning algorithms could potentially address this limitation.

test #	No. of seen processes	No. of outliers	FP rate	FN rate
test 1	2269	214	11.98%	10.76%
test 2	2221	311	13.12%	3.51%
test 3	2302	149	12.25%	3.84%
test 4	2246	260	11.92%	2.44%
average	N/A	N/A	12.31%	5.13%

TABLE III: Summary of FP and FN rates in outlier detection



(a) Seen processes (b) Outlier processes

Fig. 5: Probability difference between top two classes

B. Outlier detection accuracy

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 our 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 is learned through cross-validation, the process is classified as an outlier.

To evaluate the effectiveness of our system in identifying previously unseen processes, we repeated the experiment, this time omitting 5 randomly selected classes 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 and we applied it to the processes in the testing set. Table III summarizes the results for four different runs. For each run, we report the number of seen and outlier processes in the test set, as well as the false positive (FP) (i.e., seen process classified as outlier) and false negative (FN) (i.e., outlier classified as seen process) error rates. As may be observed, even the simple outlier screening method described above results in high outlier detection accuracy, with the average FP and FN values at 12.31% and 5.13%, respectively. This effectiveness is explained through Figure 5, which confirms our conjecture. Indeed, for previously seen processes, the probability difference between the top two classes is overwhelmingly high, while for outlier processes it is overwhelmingly low. Threshold adjustment can support biased decisions, favoring one error direction, while advanced outlier detection methods can further improve the results.

C. Logging overhead

To evaluate the overhead of our method, we focus on its major hardware component, namely storage, and we seek to assess the required data logging rate. Unfortunately, Simics is not a cycle-accurate simulator. Therefore, to attain a more accurate estimation, we calculated the logging rate 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, since the occurrence frequency can never exceed the `partition_size`. The number of partitions per second for which a vector needs to be logged 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% [20], 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 ought to be even lower in realistic cases. As a point of reference, the performance counter-based method in [17], which performs similar analysis with a different objective (i.e., malware detection vs. workload forensics), requires bandwidth of a few hundred KB/s.

VI. Conclusion

We introduced a hardware-based approach for performing workload reconstruction and process identification for the purpose of forensic analysis. Unlike OS-level and hypervisor-level methods, which rely on information obtained through the OS and are, therefore, vulnerable to software attacks, this hardware-based method extracts and logs the required information directly in hardware, making it impervious to such attacks. Herein, we demonstrated a simple incarnation of this general idea, which relies on identifying instructions causing an iTLB miss and extracting/logging appropriate features, based on which a statistical analysis can, then, perform process identification. The proposed method was evaluated on a 32-bit x86 architecture running Linux OS, which was implemented in the Simics simulation environment, alongside a statistical analysis module which employed KNN and SVM for the purpose of process classification. Experimental results using the popular Mibench benchmark suite reveal that despite the semantic gap challenge, which we addressed through the use of the CR3 register provided in x86, an overall process classification accuracy of 96.97% can be achieved at the cost of simple hardware additions capable of processing and logging data at a rate of 5.17 KB/s.

References

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

[2] AccessData. (2013) Forensic toolkit (ftk). [Online]. Available: http://accessdata.com/solutions/digital-forensics/forensic-toolkit-ftk?/solutions/digital-forensics/ftk

[3] ArsenalRecon. (2013) Registry recon. [Online]. Available: https://arsenalrecon.com/apps/recon/

[4] N.Quynh and Y. Takefuji, "Towards a tamper-resistant kernel rootkit detector," in *ACM Symp. on Applied Computing*, 2007, pp. 276–283.

[5] 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.

[6] L. Litty, H. Lagar-Cavilla, and D. Lie, "Hypervisor support for identifying covertly executing binaries," in *17th USENIX Security Symp.*, pp. 243–258.

[7] 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 Security and Privacy*, 2012, pp. 586–600.

[8] J. Pfoh, C. Schneider, and C. Eckert, "Nitro: Hardware-based system call tracing for virtual machines," in *6th International Conference on Advances in Information and Computer Security*, 2011, pp. 96–112.

[9] C. Kolbitsch, P. Milani, C. Kruegel, E. Kirda, X. Zhou, and X. Wang, "Effective and efficient malware detection at the end host," in *18th USENIX Security Symp.*, 2009, pp. 351–366.

[10] D.-Y. Yeung and Y. Ding, "Host-based intrusion detection using dynamic and static behavioral models," *Pattern Recognition*, vol. 36, pp. 229–243, 2003.

[11] J. Cabrera, L. Lewis, and R. Mehara, "Detection and classification of intrusion and faults using sequences of system calls," *ACM SIGMOD Record*, vol. 30, no. 4, pp. 25–34, 2001.

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

[13] F. Maggi, M. Matteucci, and S. Zanero, "Detecting intrusions through system call sequence and argument analysis," *IEEE Trans. on Dependable and Secure Computing*, vol. 7, no. 4, pp. 381–395, 2010.

[14] A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda, "Accessminer: Using system-centric models for malware protection," in *17th ACM conference on Computer and Communications Security*, 2010, pp. 399–412.

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

[16] 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.

[17] 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 International Symp. on Computer Architecture*, 2013, pp. 559–570.

[18] 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.

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

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