2019 IEEE 37th VLSI Test Symposium (VTS)

Hardware-based Real-time Workload Forensics via Frame-level TLB Profiling

Yunjie Zhang, Liwei Zhou and Yiorgos Makris
Electrical and Computer Engineering Department, The University of Texas at Dallas, Richardson, TX 75080, USA
E-mail:{yxz153430, lxz100320, gxm112130}@utdallas.edu

***Abstract*—We propose a hardware-based solution for performing real-time workload forensics that enables identification of a process while it is being executed. More specifically, we divide execution flow of a process into consecutive frames and we extract descriptive features related to the Translation Lookaside Buffer (TLB) utilization profile for each such frame. These features are then processed through trained machine learning models to analyze program behavior and identify workload at the granularity of a process. Unlike previous research on workload forensics that performs *ex post facto* analysis based on the *complete* process execution profile, this method *continuously* analyzes the segmented workload execution flow; thus, it does not require knowledge of process creation, switch, and termination timestamps. Furthermore, as compared with software-based workload forensics solutions, whose data logging mechanism may be compromised by software attacks, the proposed hardware-based logging mechanism does not rely on services from the operating system (OS) or high-level applications and is, therefore, inherently immune to software tampering. The proposed workload forensics method was evaluated using a Linux OS loaded on Spike, an open-source RISC-V simulator. Experimental results using the Mibench benchmark suite indicate an overall identification accuracy of 98.9% with practicable logging overhead.**

I. Introduction

As our society relies increasingly on network services and online applications, sensitive data are inevitably exposed to the threat of cyberattacks. Malware is malicious software that takes advantage of vulnerabilities in system design to bypass security policies and compromise defense mechanisms, in order to launch Denial-of-Service (DoS) attacks or steal private data. In a recent example, Spectre [1] and Meltdown [2] were able to violate the memory isolation security property and to access private information without possessing appropriate privileges. Accordingly, methods for monitoring program execution and identifying suspicious behavior are of great value.

Generally speaking, forensic analysis collects and analyzes information in order to identify or reconstruct the behavior of a program executed in the past and can be categorized into software-based methods and hardware-based methods. Numerous forensic investigation methods, such as EnCase [3] and FTK [4], have been developed in the former category. These methods utilize memory data images to analyze program control flow, while other methods focus on system call sequences to perform intrusion detection [5], [6]. However, software-based methods could be the target of a software attack themselves. For example, sensitive variables used by forensic programs are stored in a memory area which should be inaccessible by other programs; yet attacks such as the recently-developed Spectre could provide attackers a way to compromise such barrier.

On the other hand, since software has to be executed on hardware, the actual traces created during program execution cannot be hidden from the hardware. Based on that premise, the feasibility of hardware-based forensic investigation methodologies which collect data directly through hardware has also been considered [7], [8]. For instance, performance counter-based methods that monitor system events and instruction flow have been effective in on-line malware detection [8]. However, counting mechanisms which monitor the execution of every instruction and every system event result in a relatively high trace logging rate. Alternatively, the effectiveness of workload forensics which perform *ex post facto* analysis based on compacted data gathered from a complete process profile has also been investigated [7], [9]. While such methods reduce drastically the required data logging rate, they cannot respond promptly to an on-going intrusion until after a malicious processes has completed execution.

In order to address the weaknesses discussed above, we propose herein a hardware-based workload forensics methodology which (i) depends on information exclusively collected in hardware, and (ii) enables real-time process identification at any point during its execution flow and does not require knowledge of process creation, switching, and termination timestamps. To achieve these objectives, we introduce a novel approach that uses system mode switching as a flag to divide a process into separate *frames*, More specifically, upon encountering a system mode switching, a new frame is created for the execution flow of a process, thereby dividing the process into consecutive frames. Descriptive features can then be extracted for each such frame and further processed through machine learning algorithms for the purpose of process identification.

To contain the data logging overhead, we follow the successful paradigm of [9], wherein features related to Translation Lookaside Buffer (TLB) profiles are used. Moreover, besides performing the analysis in real-time rather than *a posteriori*, we also introduce a majority voting strategy which combines individual predictions generated per frame, in order to improve process identification accuracy at the expense of minor latency (i.e., a few frames rather than a single frame). The proposed method was evaluated on Spike, an open source RISC-V architectural simulator, using the Mibench benchmark suite on a Linux OS. Experimental results reveal an overall process

978-1-7281-1170-4/19/$31.00 ©2019 IEEE

identification accuracy of 98.9% based on single frame prediction and 99.7% when majority voting is used. Calculations of data processing latency and frame run-time are also provided to demonstrate feasibility of our real-time solution.

The rest of the paper is organized as follows. Related work is discussed in Section II. Section III introduces our method and provides implementation details. Section IV presents results of our experiments in process identification and outlier detection and conclusions are drawn in Section V.

II. Related Work

In this section, we briefly introduce the state-of-the-art in forensic analysis methods, including both software-based and hardware-based approaches.

A. Software-based approaches

Various data-centric software analysis methods have become standard tools for forensic investigation in industry [10]. For instance, EnCase [3], [11] ensures data integrity and enables data recovery by creating images for disk data. Other commercial tools, such as FTK [4], employ similar strategies. However, such methods exhibit limitations in processing capability, as storage capacity of electronic devices and data volumes have continued to grow rapidly over the last decade [10], [12]. Alternatively, program-centric methods focus on analyzing program behavior based on primitives such as system calls and system events. Several studies employ statistical analysis on system call sequences and/or arguments to perform intrusion detection [5], [6], [13].

B. Hardware-based approaches

Similar program-centric methods can also be found in hardware-based forensic solutions [14], [15]. They collect architecture-level information, such as CPU events, memory address references, instruction flow, etc., in order to perform malware detection and workload identification [16], [17]. For example, performance counters have been successfully used for detecting variants of malware from known malware signatures [8], [18]. Nevertheless, such ways of monitoring system events may lead to excessive complexity in CPU design and high data logging rate. In a different direction, *ex post facto* methods have proven effective in performing workload forensics and malware detection using features which are extracted and compacted from complete process execution profiles, resulting in significantly lower logging rate [7], [9]. Nevertheless, such methods are only able to detect software intrusions after an entire process has finished execution.

In addition, hardware forensics require process identification explicitly at the circuit level, without relying on any OS-level data or services. This, in turn, leads to the well-known semantic gap problem. Earlier work on process identification resolves this problem by using architectural conventions, such as the fact that the CR3 register of an x86 machine can serve as a proxy for process ID as it stores the base address of the page table of a process [19]. Moreover, any change in the value of the CR3 register corresponds to crucial events, such as process creation, switching, and termination.

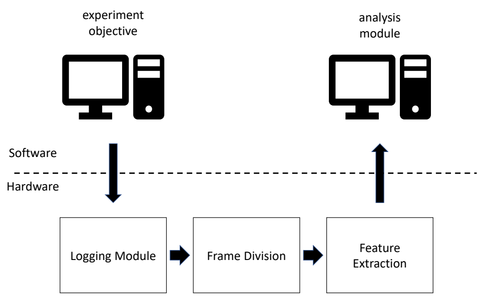


Figure 1: Overview of the proposed system architecture

III. Methodology

The objective of the proposed method is to develop a hardware-based workload forensics system that can identify a process in *real-time*, at any point during its execution. Figure 1 shows the overview of the proposed system architecture. A hardware-based logging module collects data directly from low-level hardware. To enable real-time process identification, our approach exploits a frame-division mechanism which splits the instructions into frames and continuously extracts features for each frame from the data logged by the aforementioned module. Vectors of these features are continuously off-loaded for each frame to a separate secure environment, where a software analysis module is used to perform process identification using a machine learning-based strategy which we will discuss in the following sections.

A. Logging mechanism

In order to reduce logging overhead, rather than continuously monitoring instruction execution flow, we rely on profiling instructions which cause a TLB miss. TLB in modern computer architecture is essentially a cache which stores the results of recently-used translations of virtual to physical addresses. Prior research shows that the typical TLB miss rate in software programs is about 0.01-1% [20], implying that this profiling method will induce lower logging overhead to our forensics system, as compared with performance counter-based methods which monitor and log all changes of CPU status at the instruction level. TLBs can be divided into two parts, namely instruction TLB (iTLB) and data TLB (dTLB). In our study, we only focus on the profile of instruction flow-related iTLB misses, so we discard dTLB information. In addition, our analysis module only considers user-space instructions and disregards system-mode instructions, as the former typically reflect better the program behavior-related information. In order to identify switching between user-mode and system-mode, we leverage the convention that in 64-bit Linux OS virtual addresses lower than 0x0000800000000000 are regarded as user space.

Our analysis module uses features extracted from data logged during frames occurring at any point during execution of a process. Therefore, knowledge of process creation, switching, and termination timestamps is not required. However, for the purpose of training and testing our machine learning

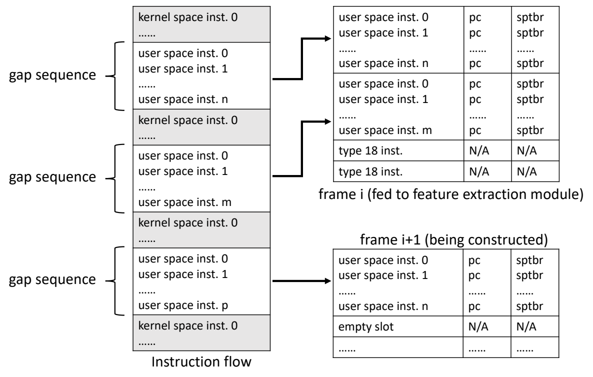


Figure 2: Frame construction in hardware

model, features need to be labeled with a corresponding process ID. Similar to the CR3 control register of the x86 architecture, RISC-V provides a register *sptbr* that holds the physical page number of the root page table and an address space identifier. This naturally offers a solution to this problem, because any change of value in *sptbr* corresponds to a context switch in the OS. By logging instructions along with their *sptbr* value, we essentially label them with a corresponding process ID, which establishes a semantic connection between hardware-level instructions and the workload to be reconstructed. The data that is logged during process execution includes three parts:

1) **iTLB miss instructions**: instructions that cause iTLB miss, including their operator and operands.
2) **sptbr**: the values seen by this register, which correspond to context switches and can be used as process IDs.
3) **program counter**: the values seen by the program counter, which can be used to distinguish user-space instructions from kernel-space instructions.

B. Frame division

Our workload forensic analysis is performed using features extracted at the granularity of a single frame. To simplify frame construction, we use a uniform size for all frames, i.e., $frame\_size$. In order to explain how workload is divided into frames, in Figure 2 we introduce the concept of $gap\ sequence$, which essentially encompasses all instructions causing a user-level iTLB miss between two kernel-to-user mode switches. Because the length of $gap\ sequence$ may vary from 1 to a large number greater than $frame\_size$, frames are constructed in a queue-like manner. The instructions of a gap sequence are pushed into the current frame if the gap sequence fits entirely into the remaining space of the frame. Otherwise, the current frame is padded with type 18 instructions (discussed in Section III-C) and passed to the analysis module and a new frame is generated. If a gap sequence is larger than the $frame\_size$, then its instructions are used to construct as many frames as possible, with completed frames passed to the analysis module, until all instructions in the gap sequence have been handled. This frame creation strategy does not require any extra information about system events or any additional capabilities of the logging mechanism.

C. Feature extraction

Feature extraction is critical for the next step of analysis, as features are expected to reflect both order and content of workload execution. Herein, we use as features the raw instruction sequences of length $frame\_size$ which cause user-level iTLB miss, as captured in each frame. Typically, a 64-bit RISC-V includes more than 200 types of operators and operands, which makes the feature space overly large for hardware implementation of the trace collection module. To address this issue, we reduce the feature space by focusing exclusively on operators and categorizing them into 18 types based on semantics given by the RISC-V specification [21]:

1) **ADD Op.**: addition operation.
2) **SUB Op.**: subtraction operation.
3) **MULT Op.**: multiplication operation.
4) **DIV Op.**: division operation.
5) **LOGIC Op.**: logic operation (AND, OR, etc.).
6) **SHIFT Op.**: shift operation
7) **LOAD Op.**: data load operation.
8) **STORE Op.** : data store operation
9) **BEQ Op.**: take branch if equal.
10) **BNE Op.**: take branch if not equal to.
11) **BGT Op.**: take branch if greater than.
12) **BLE Op.**: take branch if less than.
13) **JUMP Op.**: Unconditional jump operation.
14) **CSRR Op.**: reading CSR operation.
15) **CSRW Op.**: writing CSR operation.
16) **Floating Point ALU Op**: floating point related arithmetic or logic calculation.
17) **Floating Point DATA Op.**: floating point related data manipulation.
18) **Other Op.**: operators not included in previous types and instruction-extension related operators.

For each frame, a feature vector of $frame\_size$ $F.V._i = < Op._0, Op._1, Op._2, ..., Op._{frame\_size-1} >$ is extracted and a list of vectors $[F.V._0, F.V._1, ..., F.V._{end}]$ is collected from each process. The value of $frame\_size$ is a crucial parameter of our method. On the one hand, since we use machine learning for our analysis, a small-sized frame may have a negative impact on the overall process identification accuracy. On the other hand, a large-sized frame increases the complexity of the analysis module, runs the curse-of-dimenionality danger, and requires more hardware resources for the logged data. Thus, in our study, we seek experimentally an optimal $frame\_size$ value that can balance performance and overhead.

D. Analysis module

Our analysis consists of three separate steps, i.e., frame identification, majority voting, and outlier detection. As our workload forensics is performed at the granularity of a single frame, a basic frame identifier is required to perform multi-class classification using the features extracted from each frame, wherein each class corresponds to a single process.

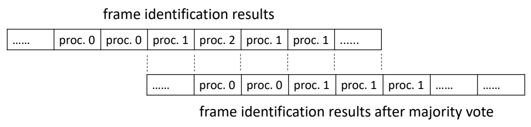

frame identification results
...... | proc. 0 | proc. 0 | proc. 1 | proc. 2 | proc. 1 | proc. 1 | ......
...... | proc. 0 | proc. 0 | proc. 1 | proc. 1 | proc. 1 | ...... | ......
frame identification results after majority vote

Figure 3: A majority voting strategy example

Similar classification tasks can be found in research dealing with word sequences, such as machine translation and speech recognition. Two types of Recurrent Neural Networks (RNNs) with gating units, namely Long Short-Term Memory (LSTM) and Gated Recurrent Units (GRUs), have been developed to address these tasks. While there is no significant difference observed in the performance of these two models, GRUs-RNN may be slightly advantageous in computational time [22]. As a result, the latter model is used in our analysis module.

In order to further improve the accuracy of our forensics method beyond the abilities of a single frame identifier, we employ majority voting after several frames have been processed. Since process identification is taking place in real time using a large number of successive frames, this approach assists with suppressing sporadic frame prediction errors through the decisions of neighboring frames, at the expense of a small latency. An example is given in Figure 3, where a prediction is given after 3 frames have been processed. In case of a tie, our method picks the earliest frame identification. Additionally, frames from unseen processes can be identified through outlier detection. We leverage the fact that the sigmoid output layer of our neural network returns a vector of probabilities of frame classes. If a frame is from a process which has never been seen in training, its prediction may not generate a dominating likelihood in any one of the target classes. Herein, we utilize this property of ambiguity to perform outlier detection. In our experiments, we identify a minimum likelihood threshold that needs to be exceeded in order to classify a frame as seen.

IV. Experimental Results

In this section, we evaluate our process identification method as well as its data logging overhead. Our experiments are conducted on Spike, a RISC-V simulator configured to work with the RISC-V 64-bit instruction set, on which we install a 64-bit Linux 2.6 OS kernel with necessary applets.

A. Process Identification Performance

As workload for our experiments we select the MiBench suite. However, due to the on-going development of the RISC-V compiler library, some MiBench applications cannot be compiled properly and, thus, are not included in our experiments. Applications are executed with various arguments and in random order so as to eliminate any possible bias introduced by program execution order. Since the source code of Spike is available online, it provides us with great flexibility for natively implementing the data logging module and the feature extraction module. Here, we embed an iTLB tracer in the MMU class of the Spike simulator, in order to track TLB accesses. Each time an iTLB miss occurs, it raises a flag which prompts another modified function to log the user-level instructions that cause iTLB misses along with the corresponding program counter (PC) values before proceeding with instruction execution. Also, in order to gather data for calculating the needed logging rate, a counter is created in the simulator to record the total number of instructions executed while each frame is created. Data analysis and result evaluation are performed with TensorFlow on Python 3.6.

Table I: Frame-level process identification results on RISC-V

application class	**training samples**	**testing samples**	**GRUs accuracy**
overall	34367	24879	98.9%
crc	2557	1705	99.6%
fft	3225	2150	99.2%
qsort	3125	2084	99.4%
toast	3624	2428	99.4%
search	3676	2450	99.4%
bitcnts	2784	1864	98.7%
untoast	2670	1782	99.4%
dijkstra	2538	1692	98.4%
patricia	2857	1904	99.3%
basicmath	2934	1956	96.0%
bf	3576	2384	100%
susan	3704	2480	98.0%

In order to evaluate the accuracy of frame-level process identification, we collected a dataset containing iTLB miss profiles from a total of 71 processes. In this initial experiment we used a $frame\_size$ of 30 (as we discuss later, this is the experimentally-observed optimal value) and observed 59,246 frames generated by our frame division strategy, each of which was labeled with a corresponding process ID. For each process, 60% of the samples were randomly selected as our training set while the rest were used as our testing set. The process identification results are shown in Table I. As may be observed, the GRUs-RNN exhibits excellent process identification performance, reaching an overall accuracy of 98.9%. Interestingly, these results reveal that our method not only matches but also outperforms previous *ex post facto* analysis methods such as [9], despite operating in *real time* and utilizing a small fraction of the trace used by such methods.

To explore the impact of $frame\_size$ on process identification accuracy, we conducted the same experiment while varying the value of $frame\_size$ between 16 and 34. The results are shown in Figure 4, revealing that the overall process identification accuracy starts to improve significantly once $frame\_size$ exceeds 22, yet flattens out once $frame\_size$ exceeds 30. Therefore, a $frame\_size$ value of 30 is adopted as our experimentally-observed optimal value.

B. Majority Voting Results

With the above results as our baseline reference point, we proceeded to evaluate the added effectiveness that may be obtained by employing a majority voting strategy across multiple frames, rather than relying on a single frame for identifying a process. To this end, we used the collected in-order execution profile and applied this strategy for a range of different latency values. Latency here refers to the number of frames that we rely on for making a majority-based decision.

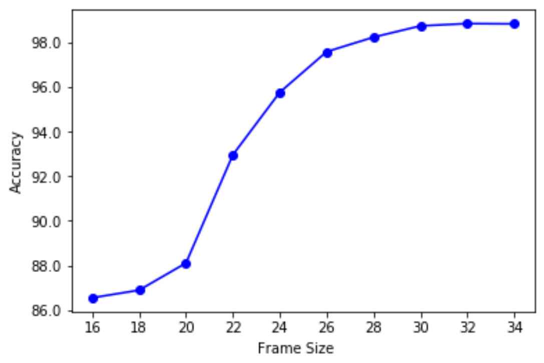


Figure 4: Process identification accuracy vs. frame size

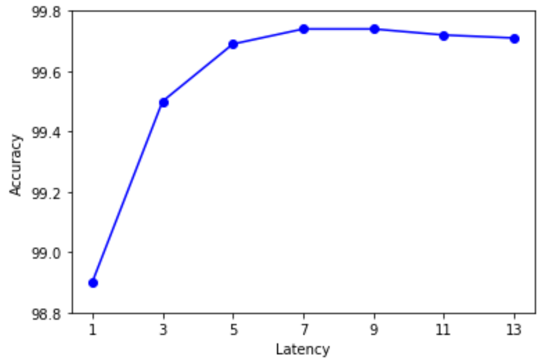


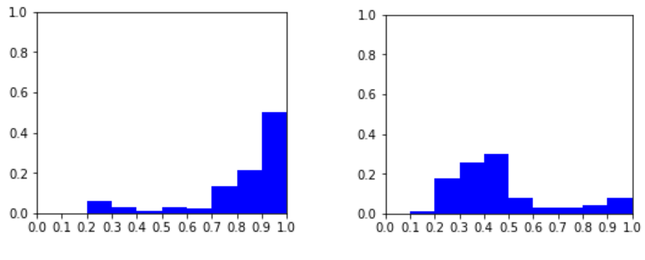
Figure 5: Process identification accuracy vs. latency

As shown in Figure 5 for the chosen $frame\_size$ of 30, when latency n is set to 7 the overall process identification accuracy improves significantly, rising from 98.9% to 99.7% once majority voting is applied.

C. Outlier Detection

In order to detect outlier (i.e., previously unseen) processes, we utilize the information provided by GRUs-RNN with a sigmoid output layer, which reflects the likelihood that a given input stimulus (i.e., frame) belongs to certain known classes. As we observed experimentally, for frames originating from seen classes (i.e., processes) the likelihood that they belong to the predicted class comes with a strongly dominating value. However, for frames originating from previously unseen classes, no class exhibits a dominant likelihood. An example of likelihood distribution is shown in Figure 6 for each of the above two cases. Based on this observation, we can screen outlier processes by setting a minimum threshold for the highest probability in the GRUs-RNN's output vector. Any frame for which no class likelihood exceeds this threshold can, thereby, be deemed as not belonging to a known process.

We conducted multiple iterations of our experiment, each time randomly selecting and excluding 25% of the classes (i.e., processes) in the training set, while retaining them in the validation set, thereby making them outliers (i.e., previously unseen). As a threshold for accepting the highest likelihood class of our GRUs-RNN, we selected 0.65, as this value gave



(a) Seen processes (b) Outlier processes

Figure 6: Probability distribution of top classes

Table II: Summary of FP/FN rates in outlier detection

test #	No. of seen frames	No. of outlier frames	FP rate	FN rate
average			14.47%	7.95%
test 1	19501	5378	15.74%	6.96%
test 2	18137	5146	14.55%	8.46%
test 3	18940	5939	11.26%	9.90%
test 4	17632	6261	18.38%	5.57%
test 5	18097	6782	12.41%	8.87%

us experimentally the highest average F1 score[1]. Results from 5 random iterations with threshold 0.65 are summarized in Table II. In our case, outliers are defined as the positive class and the table provides the false positive rate (seen processes classified as outliers) and false negative rate (outliers classified as seen processes). As may be observed, this straightforward method of outlier screening achieves reasonably accurate results, with average FP and FN rates of 14.47% and 7.95%, respectively. While it is possible that more advanced machine learning models may offer better outlier detection accuracy, our method relies only on the existing analysis model and does not require additional processing. This is particularly important as any added overhead could jeopardize our ability to perform this analysis in real-time and our future research direction of implementing the entire workload forensics method on-chip.

D. Logging Rate

The amount of data per second that needs to be passed to the analysis module for processing is referred to as the logging rate. This metric can be used to describe the overhead introduced by the logging and feature extraction modules. High logging rate requires higher processing speed and potentially a storage buffer if collected data cannot be processed at the rate of arrival. To compute the logging rate of our frame-level analysis, which extracts a feature vector of length $frame\_size$, we assume a cycles-per-instruction (CPI) value of 1 and we introduce the following variables:

$$Feature\ Size = log_2(\#\ of\ Op\_types) \times\ frame\_size \quad (1)$$

$$Logging\ Ratio = Feature\ Size \times Frames\ per\ Cycle \quad (2)$$

$$Logging\ Rate = Frequency \times Logging\ Ratio \quad (3)$$

In our experiments, the average number of executed cycles while constructing a frame was 3.6×10^5. For a $frame\_size$

[1]The F1 score is the weighted harmonic mean of the precision and recall of the GRUs-RNN classification.

Table III: Analysis time summary

frame time (*ms*)	**identification** (*ms*)	**majority vote decision** (*us*)	**transmission delay** (*us*)
0.277	0.0679	1.46	less than 10

of 30, we can then calculate that $Frames\ per\ Cycle$ is 2.78×10^{-6}. Assuming a 1.3 GHz clock frequency, as reported in the RISC-V processor prototype of [23], our logging rate is estimated to be about 66.1KB/sec, which is significantly lower than that of performance counter-based methods [8].

E. Analysis Latency

In order to perform continuous process identification, the time required for analyzing the collected data should not exceed the time for constructing a frame, i.e., $frame\ time$. Despite relying on an analysis module implemented in software and running on a separate system where the logged traces are passed to and analyzed by, real-time processing is still feasible. Even without the use of any custom hardware accelerator or GPU optimization, the time needed for analysis is much less than $frame\ time$, as we explain below using the parameters given in the previous section. Our average analysis time is summarized in Table III and includes the average time for frame construction, data transmission, frame-level process identification and majority voting latency. The data transmission delay is estimated based on the results described in [24], where less than 10*us* of one-way latency is introduced at a bandwidth of 2.1Gbps if TCP/IP ethernet is applied.

In our experiment, it took an average of 11.3 ms to obtain the first frame-level process identification. At the same time, the average time to complete execution of a process was 2.43 seconds, while the shortest process run time was 225 ms. Evidently, even for the shortest running process, our method provides results in an order of magnitude faster time, thereby corroborating feasibility of real-time process identification.

V. Conclusion

In this work, we explored the feasibility of performing hardware-based real-time workload forensics. Unlike software-based approaches, the proposed method is immune to software attacks as it does not involve data collected from the OS or software applications. We extract features from instructions causing an iTLB miss directly through hardware and we construct frames which are then analyzed further through trained machine learning models in order to identify the running process. We demonstrated our method on a modified version of a RISC-V architectural simulator, i.e., Spike, running the Mibench benchmark suite on a 64-bit Linux OS. An overall process identification accuracy of 98.9% is achieved when frames of 30 user-level instructions causing iTLB miss are used as features, while even higher accuracy of 99.7% is achieved when a majority voting strategy is employed for successive frames. Finally, we also demonstrated that the time required for collecting and analyzing the traces is sufficiently low for performing real-time workload forensic analysis.

References

[1] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," *arXiv:1801.01203*, 2018.

[2] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown," *arXiv:1801.01207*, 2018.

[3] S. Bunting and W. Wei, *EnCase Computer Forensics: The Official EnCase Certified Examiner*, 2006.

[4] AccessData, "Forensic toolkit," 2013. [Online]. Available: https://accessdata.com/products-services/forensic-toolkit-ftk

[5] D.-Y. Yeung and Y. Ding, "Host-based intrusion detection using dynamic and static behavioral models," *Pattern Recognition*, vol. 36, no. 1, pp. 229–243, 2003.

[6] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. Wang, "Effective and efficient malware detection at the end host." in *USENIX Security Symposium*, vol. 4, no. 1, 2009, pp. 351–366.

[7] L. Zhou and Y. Makris, "Hardware-based workload forensics and malware detection in microprocessors," in *International Workshop on Microprocessor and SOC Test and Verification (MTV)*, 2016, pp. 45–50.

[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 *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, 2013, pp. 559–570.

[9] L. Zhou and Y. Makris, "Hardware-based workload forensics: Process reconstruction via tlb monitoring," in *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2016, pp. 167–172.

[10] D. Ayers, "A second generation computer forensic analysis system," *Digital Investigation*, vol. 6, pp. S34–S42, 2009.

[11] S. Widup, *Computer forensics and digital investigation with EnCase Forensic v7*, 2014.

[12] D. Quick and K.-K. R. Choo, "Impacts of increasing volume of digital forensic data: A survey and future research challenges," *Digital Investigation*, vol. 11, no. 4, pp. 273–294, 2014.

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

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

[15] L. Zhou and Y. Makris, "Hardware-assisted rootkit detection via on-line statistical fingerprinting of process execution," in *Design, Automation & Test in Europe Conference (DATE)*, 2018, pp. 1580–1585.

[16] M. Ozsoy, K. N. Khasawneh, C. Donovick, I. Gorelik, N. Abu-Ghazaleh, and D. Ponomarev, "Hardware-based malware detection using low-level architectural features," *IEEE Transactions on Computers*, vol. 65, no. 11, pp. 3332–3344, 2016.

[17] L. Zhou and Y. Makris, "Hardware-based on-line intrusion detection via system call routine fingerprinting," in *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2017, pp. 1550–1555.

[18] A. Tang, S. Sethumadhavan, and S. J. Stolfo, "Unsupervised anomaly-based malware detection using hardware features," in *International Workshop on Recent Advances in Intrusion Detection*, 2014, pp. 109–129.

[19] S. T. Jones, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau *et al.*, "Antfarm: Tracking processes in a virtual machine environment." in *USENIX Annual Technical Conference*, 2006, pp. 1–14.

[20] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design*, 2007.

[21] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovic, "The RISC-V instruction set manual, volume i: Base user-level ISA," *University of California*, 2011.

[22] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," *arXiv preprint arXiv:1412.3555*, 2014.

[23] Y. Lee, A. Waterman, R. Avizienis, H. Cook, C. Sun, V. Stojanović, and K. Asanović, "A 45nm 1.3 GHz 16.7 double-precision gflops/w risc-v processor with vector accelerators," in *40th European Solid State Circuits Conference (ESSCIRC)*, 2014, pp. 199–202.

[24] S. Larsen and B. Lee, "Survey on system I/O hardware transactions and impact on latency, throughput, and other factors," in *Advances in Computers*, 2014, vol. 92, pp. 67–104.