TPE: A Hardware-Based TLB Profiling Expert for Workload Reconstruction

Liwei Zhou, *Member, IEEE*, Yunjie Zhang, *Graduate Student Member, IEEE*, and Yiorgos Makris, *Senior Member, IEEE*

***Abstract*— We propose TPE, a hardware-based framework to perform workload execution forensics in microprocessors. Specifically, TPE leverages custom hardware instrumentation to capture the operational profile of the Translation Lookaside Buffer (TLB), as well as process these information off-line through multiple machine learning and/or deep learning approaches, in order to identify the executed processes and reconstruct the workload. Unlike software-based forensics methods implemented at the operating system (OS) or hypervisor level, whose data logging and monitoring mechanisms may be compromised through software attacks, TPE is implemented directly in hardware and, therefore, provides innate immunity to software tampering. A prototype of TPE is demonstrated in Linux on two representative architectures, i.e., 32-bit x86 and 64-bit RISC-V, implemented in the Simics and Spike simulation environment respectively. Experimental results using the Mibench workload benchmark suite reveal favorable process identification accuracy at low logging rate, which corroborates the effectiveness and the generalizability of TPE.**

***Index Terms*— Workload forensics, TLB, machine learning, hardware-based system security, RISC-V.**

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.

Generally speaking, traditional computer forensics analysis focuses on interpreting critical OS or application data structures in binary form as human-understandable semantics. Specifically, static pattern matching approaches are widely adopted in order to retrieve or infer the data structures of interest from the memory image, while human expertise is typically required to bridge these data structures with the events that transpired in further analysis [1], [2]. Numerous such methods have been developed at the OS level, leveraging the convenience and flexibility of memory image probing and analysis implementation [3]–[5]. OS-level methods, however, can be subject to software attacks staged at the same level. Kernel rootkits, for example, may compromise the OS-level logging system and eliminate all traces associated with malicious actions.

In order to address this limitation, hypervisor-level forensics solutions have been proposed [6]–[8]. A hypervisor 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, the data collected by forensics methods at the hypervisor level is generally immune to OS-level software attacks. Nevertheless, as shown through recent work [9], the hypervisor itself can be the attack target, as several vulnerabilities and intrusion methods have been identified. As a result, similar attacks can be launched at the hypervisor, which compromise the integrity of the data acquired at this level, in order to conceal malicious events and subvert the effectiveness of the forensic analysis.

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. As a result, there exists no physical pathway for the OS, hypervisor, or any application running on the system to interfere with the logged data. Accordingly, traces obtained from hardware are expected to be immune to software-based tampering. On the other hand, 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 forensics solution, i.e., TPE, which seeks to reconstruct executed workload at the granularity of a single process, through minimal information obtained from the Translation Lookaside Buffer (TLB). A preliminary exploration of this idea was presented in [10]. This extended version, however, explores several additional aspects and provides a more comprehensive exposition of the proposed method, as summarized below:

1) **Architecture-agnostic approach:** Two representative architectures, i.e., 32-bit x86 and 64-bit RISC-V, are evaluated with the TPE. Similar results are obtained

Manuscript received December 7, 2020; revised March 4, 2021; accepted April 16, 2021. Date of publication May 4, 2021; date of current version June 14, 2021. This article was recommended by Guest Editor J. M. Fung. *(Corresponding author: Yunjie Zhang.)*

Liwei Zhou is with Google, Mountain View, CA 94043 USA (e-mail: zlw_frank110@hotmail.com).

Yunjie Zhang and Yiorgos Makris are with the Department of Electrical and Computer Engineering, The University of Texas at Dallas, Richardson, TX 75080 USA (e-mail: yxz153430@utdallas.edu; gxm112130@utdallas.edu).

Color versions of one or more figures in this article are available at https://doi.org/10.1109/JETCAS.2021.3077442.

Digital Object Identifier 10.1109/JETCAS.2021.3077442

from both platforms, indicating that TPE is effective across architectures.

2) **Advanced analysis method:** Compared to the work in [10], more advanced machine learning algorithms are evaluated and their impact on the accuracy of the process identification and the outlier detection tasks are assessed.
3) **Advanced feature mining:** The features used for process identification and outlier detection are further refined and more feature combinations are explored. Specifically, the experimental results reveal (1) the impact of the granularity of the feature space on the accuracy of the process identification and the outlier detection, and (2) the effectiveness of early prediction, which significantly reduces the feature logging cost, and supports the possibility of a real-time solution.

The remainder of the paper is structured as follows. In Section II, we discuss related work. The system model of the TPE is introduced in Section III. Section IV, Section V and Section VI present details of our logging and feature extraction mechanism as well as the corresponding analysis methods. Experimental results evaluating the accuracy of the TPE in reconstructing workload, as well as its overhead, are presented in Section VII. The hardware implementation details are provided in Section VIII. Potential limitations are discussed in Section IX, while conclusions are drawn in Section X.

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, hypervisor-level and hardware-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 [11]. Table I provides a taxonomy of all related methods described in this Section, including the method proposed in this paper.

A. OS-Level Approach

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 may focus on the integrity of file system objects, such as files on disks. Various commercial products fall into this paradigm. For example, enCase creates images for disk data to enable data recovery and/or to ensure data integrity. Similar products include FTK [1] and Registry Recon [2]. Alternatively, integrity of control flow of code, such as kernel services is another hotspot in data-centric approaches. In particular, Control Flow Integrity (CFI) checking, which generally splits the code execution flow into different chunks at different granularities and attests the correctness of the real-time chunk-based execution chain, has been proposed as a promising defense against control flow hijacking attacks such as Code-Reuse Attacks (CRA). However, the incurred implementation and/or runtime overhead is relatively high [3]–[5].

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 malware detection follows this paradigm [12]–[14]. In general, these methods rely solely on analysis of system call sequences. An interesting extension is introduced in [15], 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 [16], 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 Approach

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 [17] 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 [6], [18]. 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 [19]. Such methods even go to a lower layer, to check whether contents on the disk and its image in main memory match [7], [8]. Nevertheless, they still rely on OS-level information (e.g. `system.map`) to locate which part is critical to keep their eyes on [8].

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 [20]. A more recent approach, therefore, investigates the use of performance counters to model the system call execution flow in order to detect kernel rootkits, which may intentionally contaminate the execution behavior [21].

C. Hardware-Based Approach

Hardware-based approaches are generally immune to software attacks, thereby enabling a new direction toward

TABLE I
TAXONOMY OF RELATED WORK AND TPE

	Data-centric	Program-centric
OS-level	[1], [2], [3], [4], [5], [22]	[12], [13], [14], [15], [16]
Hypervisor-level	[7], [8], [19], [21]	[6], [17], [18]
Hardware-level	[22], [23], [24], [25]	[10], [26], [27], [28], [29], [30], [31], [32], [33], [34], **TPE**

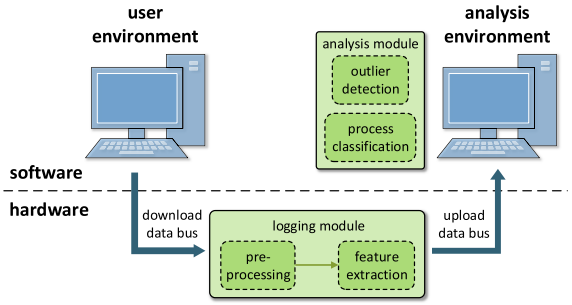


Fig. 1. High-level system architecture.

thwarting malicious software [35]. For instance, similar software-based data-centric methodologies, e.g., CFI checking methodologies, can be applied at hardware level, which are expected to incur less runtime overhead [22]–[24], [36]. In contrast to traditional CFI methods, CFIMon uses performance counters to model code execution behavior and detect control flow deviation [25]. Nevertheless, these methods generally require specific support from the underlying OS or compiler to bridge the semantic gap [22], [23].

While system call-related information can also benefit hardware-based approaches in malware detection [26], [27], most of program-centric 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. For instance, performance counter can be used to model the program behavior through machine learning methods, based on which malware detection can be performed [28], [30]–[32]. Besides performance counter, alternative methods collect low-level architectural information, e.g. memory address reference, instruction opcode, etc., to model program behavior and perform malware detection [29], [37]. Furthermore, a preliminary version of this work, which performs workload reconstruction through Translation Lookaside Buffer (TLB) profiling has been proposed in [10]. Based on the successful paradigm of [10], TLB profiling has also been applied in real-time process identification tasks [33], [34].

III. TPE OVERVIEW

The primary objective of the TPE is to develop a system-level workload reconstruction capability at the granularity of process through data collected exclusively from the hardware. Specifically, the TPE models the program behavior based on the runtime architectural events using machine learning algorithms, in order to identify if a process is known or not and what application a process actually is. A top-level view of the TPE architecture is shown in Figure 1. In particular, the TPE consists of two main components, namely *logging module* and *analysis module*. The logging module, which resides in the hardware to prevent software tampering, continuously logs the data required for program behavior modeling and extracts descriptive features from the collected data. A dedicated port, which is invisible to and inaccessible by the OS, is then involved to off-load the extracted features into a trusted software environment. Accordingly, the analysis module, which implements machine learning algorithms, can then perform process classification as well as outlier detection in order to reconstruct the executed workload.

The TPE is evaluated on two OS/architecture platforms, i.e., 32-bit Linux/x86 and 64-bit Linux/RISC-V. A modern computer architecture design falls into the category of either a Complex Instruction Set Computing (CISC) architecture or a Reduced Instruction Set Computing (RISC) architecture. Correspondingly, the x86 architecture is a representative CISC architecture, which is widely adopted in Intel microprocessor family. On the other hand, the RISC-V architecture is an open-source representative RISC architecture, which was initially developed by the University of California, Berkeley, and provides extensive flexibility for various industrial or research purposes. Furthermore, a modern OS can adapt itself to a 32-bit version or a 64-bit version, according to different architecture support. Consequently, the evaluation on both the OS/architecture platforms ensures the practicality and generalizability of the TPE.

IV. LOGGING OBJECT

Hardware-based forensics approaches, similar to hypervisor-level approaches, suffer from the semantic gap problem. Specifically, in this work, to model the program behavior at the architecture level, there are two questions to be addressed: (1) what data in hardware could be collected as the identifier of a process? (2) what data in hardware could be collected to describe the behavior of a process? In this Section, we review the potential features of x86 and RISC-V architecture which are beneficial for bridging the corresponding semantic gap.

A. x86 Architecture

1) Process Identifier: In modern OS, thanks to the concept of virtualization, each process has its dedicated address space, consisting of continuous virtual memory blocks, that maps resources used by the process into physical memory. This mapping is fulfilled by the translation between virtual address (i.e., addresses to access virtual memory) to physical address (i.e., addresses to access physical memory), which is maintained by a dedicated page table. As a result, the base address of the page table can be accurately bound with a process. In x86, this base address is stored in a control register CR3, whose value-changes perfectly match the events of process creation, switching and termination [17]. Therefore, in this work, we use the CR3 value as the identifier of a process.

2) Process Behavior: Program execution behavior typically follows phases, which can be effectively predicted via performance counter values. Modern microprocessors, with the assistance of the OS kernel and the hyperthreading technique, can even provide fine-grained per-thread data for more accurate

process behavior modeling. Unfortunately, the engagement of the OS kernel expands the attack surface. Moreover, order of program execution will affect performance counter values. As a result, associating these values accurately with OS-level objects, such as processes, is not at all straightforward.

To address this limitation, rather than using performance counter values, our approach uses instructions causing TLB misses to profile the process behavior. 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 32-bit 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).

B. RISC-V Architecture

1) Process Identifier: The RISC-V architecture also follows the principle of page virtualization and maintains a page table which facilitates the translation between virtual addresses and physical addresses. Accordingly, the base address of a page table, whose value is stored in the Supervisor Page-Table Base Register (SPTBR), can be used as the process identifier to track the currently-active process.

2) Process Behavior: In RISC-V architecture, unlike x86, the simulated TLB model contains only iTLB while ignoring the dTLB implementation. Nonetheless, this fact does not affect the use of user-space instructions raising iTLB to describe the process behavior. Furthermore, since the iTLB is also flushed every time when a SPTBR value is changed, our approach still shares the same benefits of the design convention as in the case of x86. In the 64-bit Linux OS, the user space and the kernel space are separated by a huge void space. As a result, all iTLB miss events incurred by virtual addresses higher than `0xFFFFFF0000000000`, which point to the kernel space, are ignored in this case. Similarly, in the end, each SPTBR value is associated with the corresponding sequence of instructions which incurred iTLB misses.

V. Feature Extraction

In order to model the process behavior using machine learning algorithms, descriptive features have to be extracted from the logged data, i.e., sequences of user-space instructions raising iTLB misses. To minimize the data logging overhead, our feature extraction is also performed directly in hardware. As a result, the selected features are expected to be sufficiently descriptive while the feature exaction mechanism itself should not be too complicated and incur impractical hardware design overhead. To simplify the procedure of feature extraction, we first pre-process the logged data to obtain an abstraction of its semantics. Upon the pre-processed data, three types of features, including two *independent features* and one *sequential feature*, are then developed and evaluated. The same methodology is shared between the x86 and RISC-V architecture, while a slight difference is involved in data pre-processing due to the distinction between the x86 instruction set and RISC-V instruction set.

A. Pre-Processing Data

1) Pre-Processing on x86: We first abstract the semantic of the logged instruction sequence through categorizing the operator and operand of instructions. Six types of operators (Op.) are considered on x86 as follows:

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.

We consider 12 categories of operands (Opr.), including 8 classes corresponding to the 8 general purpose registers, 1 for memory reference, 1 for XMM registers and floating point stack, 1 for all segment registers, and 1 for immediate value. Upon these 18 types of Op./Opr., the exact features representing the process behavior on x86 can then be developed.

2) Pre-Processing on RISC-V: Unfortunately, dedicated **Stack Op.** and **I/O Op.** are not available in RISC-V instruction set, and thus, the same classification of operators cannot be directly applied in this scenario [38]. Therefore, these two categories are excluded. Furthermore, the RISC-V implements a group of dedicated instructions manipulating the Control and Status Registers (CSR) to facilitate program execution. CSRs manage various common CPU tasks, e.g., interrupt and exception handling, paging switch and addressing, etc., as well as maintain the status of the process and the flags raised by different program executions. Therefore, a new category, i.e., **CSR Op.**, is included in the classification, which results in the following five types of operators:

1) **Data Op.**: same definition as in x86.
2) **ALU Op.**: same definition as in x86.
3) **Control Flow Op.**: same definition as in x86.
4) **Floating Point Op.**: same definition as in x86.
5) **CSR Op.**: operations manipulating CSR register family.

Additionally, 13 categories of operands (Opr.) are considered herein. These include 1 class for stack pointer, 1 for global pointer which tracks access to the heap, 1 for thread pointer which points to thread-local storage, 1 for program counter and 1 for immediate value. Moreover, 4 classes are considered for function call-related operands, i.e., the registers which hold the return address, the temporary registers which hold intermediate results during function execution, the saved registers which hold the values that should be maintained across function calls, and the registers for function arguments and return values. Another 4 counterparts are considered for function calls involving floating point arithmetic. Upon these 18 types of Op./Opr., the exact features representing process behavior on RISC-V architecture can be developed.

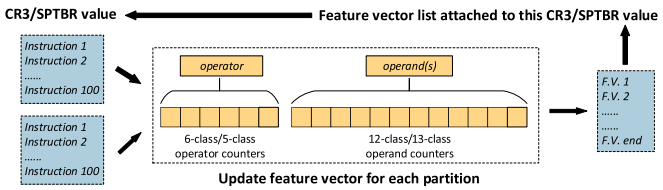


Fig. 2. Feature extraction - counts of occurrence.

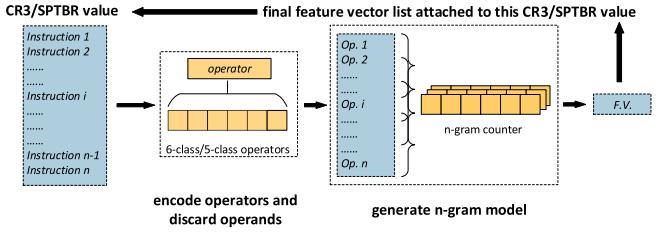


Fig. 3. Feature extraction - n-gram.

B. Counts of Occurrence With Partitioning

The first feature developed herein is the counts of occurrence of the categorized Op./Opr. Conceptually, for each process, its associated instructions causing iTLB misses are first partitioned into sets of a maximum size of `partition_size`. Partitioning helps retain order information while reducing log size. In one extreme, choosing `partition_size` to be one 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 TPE, we experimented with `partition_size` of 100 instructions.

A vector $F.V._i = < Op._1, \ldots, Op._n, Opr._1, \ldots, Opr._m >$ is extracted for each partition. For each process, as identified through its CR3/SPTBR value, a list of feature vectors $[F.V._1, \ldots, F.V._i, \ldots, F.V._{end}]$ 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.

Counts of occurrence benefits from its simplicity of implementation. The partitioning, on the other hand, provides finer-grain view of counts of occurrence as well as reflects the order information of instruction sequence, though in a lossy way. We also note that, because of the variant number of possible partitions in one process, which can vary from hundreds to thousands in our scheme, the size of the generated feature matrix can be extremely large so that dimension reduction methods must be applied in the further forensic analyses using machine learning, otherwise they may suffer from the curse of dimensionality problem while the computational overhead of the analysis may also be dramatically increased.

C. n-Gram Model

An alternative but popular way of feature extraction is using the n-gram model. A *n-gram* is a subsequence of n items derived from a given sequence. A feature matrix can then be constructed by the number of multiple possible n-gram subsequences. Therefore, similarly, when n is greater than two, n-gram model can also preserve both frequency and order information, while the order information is less lossy with larger n. The n-gram model is advantageous in the size of the feature matrix, since the total number of features can be fixed and bound by the number of possible elements in a given sequence m and the choice of n, i.e., m^n. However, n-gram model is generally applied on an univariate sequence; therefore, it cannot be directly used on our logged instruction sequences, which are multivariate sequences containing variables of both operators and operands.

Current research on program behavior modeling generally only adopt features generated from sequence of 'operators', without considering 'operands' (e.g., relying on sequence of system call number solely and ignoring its arguments [12]–[14]) while remaining effective. Similarly in our scheme, the operators can be expected to convey more information of program behavior than the operands do. Therefore, we discard the operand part in our logged instruction sequence while retain only the operator part so that the original multivariate instruction sequence can then be converted into a univariate operator sequence, on which the n-gram model can be easily applied, as shown in Figure 3. As a result, feature extraction using n-gram model, as compared with number of occurrence with partitioning, maintains frequency and lossy order information with a feature matrix of significantly reduced size, whereas it ignores operand information.

D. Raw Sequence of Categorized Operator

Essentially, both feature extraction methods introduced in Section V-B and Section V-C try to extract significant features from a lossy compression of the logged instruction sequence while these features are handcrafted, requiring human intelligence and/or experience. Inevitably, there is no guarantee that the selected features are the most representative while some descriptive information, e.g., the precise order information, may also be accidentally filtered due to the compression. As a result, crafting features, which capture both frequency and order information more precisely, may be beneficial. To this end, we employ the entire raw instruction sequence, without any further feature extraction, as the feature vector. Indeed, the original sequence is able to maintain the lossless frequency and order information. However, traditional machine learning methods, which expect *independent features* in the feature vector, such as the features introduced in the previous Section, cannot accept sequential inputs. Hence, it is necessary to employ more advanced machine learning algorithms, such as deep learning models, to process the *sequential features*.

Deep learning is a branch of machine learning which attempts to model high-level abstraction of data through multiple processing layers. Using deep learning models benefits us in two ways. Firstly, certain architectures of the deep learning model can process sequential inputs so that they are a perfect fit for our sequential features. Secondly, deep learning algorithms can mine representative features from the raw sequence automatically, without human intervention; thereby, optimal features may be generated. Details of the exact learning model we apply will be explained in the following Section.

Due to the fact that existing deep learning models limit their capability to processing only univariate sequences, as well as the assumption that operators are more informative in program

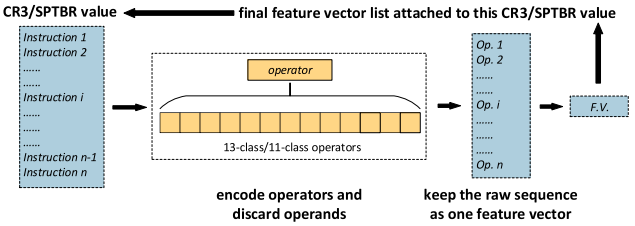


Fig. 4. Feature extraction - raw operator sequence.

TABLE II
EXTENDED CLASSES OF DATA OP./ALU OP. SET FOR BOTH X86 AND RISC-V SCENARIO

Original class	Extended class	Description
DATA Op. (x86)	DATA Op.	data manipulation operation
	ADDR Op.	address manipulation operation
	FLAG Op.	flag manipulation operation
DATA Op. (RISC-V)	LOAD Op.	data load operation
	STORE Op.	data store operation
ALU Op. (x86 & RISC-V)	ADD Op.	addition operation
	SUB Op.	subtraction operation
	MULT Op.	multiplication operation
	DIV Op.	division operation
	LOGIC Op.	logic operation (AND, OR, etc.)
	SHIFT Op.	shift operation

behavior modeling, the actual feature we use is the operator sequence while operand information is discarded, as shown in Figure 4. Furthermore, after examining the log of instruction sequences, it is revealed that most instructions fall into **DATA Op.** set and **ALU Op.** set. Hence, for both the x86 and the RISC-V scenarios, we extend the categories of these two operators as illustrated in Table II. Consequently, 13 classes of operators, rather than the original 6 classes, are evaluated for x86, while 11 classes of operators, rather than the original 5 classes, are evaluated for RISC-V.

E. Early Prediction

State-of-the-art program-centric forensics approaches generally model the program behavior through profiling their entire execution flow. However, it has been revealed that most program behaviors tend to diverge at an early stage so that a subsequence of their execution flow can be sufficient to provide distinguishable features [26]. In this work, we evaluate the *early prediction* effect on features introduced in Section V-C and Section V-D. Specifically, we perform the corresponding feature extraction mechanism only on a fixed-length subsequence of the operator sequence. Various possible lengths of the subsequence are experimented with exhaustively while the optimal length is selected based on statistical observations. Apparently, the *early prediction* effect simplifies the feature extraction mechanism in hardware, leading to significant decrease in memory overhead, as well as reduction in the size of feature matrices so that the computational complexity of the follow-on forensics analysis can be reduced.

VI. MACHINE LEARNING FOR FORENSICS

The objective of the analysis module is to reconstruct workload execution at the granularity of a process using the extracted features. Since forensics is typically an *ex post facto* effort, the actual analysis is implemented 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 [37]. 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 note that the methodology for forensics analysis is *architecture-agnostic*.

A. Process Reconstruction

For the purpose of process reconstruction, we experimented with three different non-linear multi-class classifiers of varying complexity and performance, namely K-Nearest Neighbors (KNN), Support Vector Machine (SVM) and Recurrent Neural Network (RNN). KNN and SVM, as traditional machine learning algorithms, are employed to handle the *independent features*, i.e., the features introduced in Section V-B and Section V-C, while RNN, as the more advanced deep learning algorithm, is employed to handle the *sequential features*, i.e., the feature introduced in Section V-D.

1) Handling Independent Features: 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, using counts of occurrence and n-gram model, 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 built-in library and SVM from the LIBSVM library [39].

RNN, as a variation of the traditional Artificial Neural Network (ANN), has been developed in order to make use of sequential information of the input. The traditional ANN has a unidirectional multi-layer structure, where each layer consists of a user-defined number of nodes, i.e. *neurons*, which are interconnected with nodes in the adjacent layers. The leftmost layer of the network is generally termed *input layer* while the rightmost layer is termed *output layer*. All the intermediate layers, on the other hand, are termed *hidden layers*. Each neuron in a hidden layer is then composed of a tunable parameter matrix W, called *weight* as well as a user-defined function F, called *activation function*, which performs the mapping $y = F(W^T x)$, where x and y are the corresponding input and output of the neuron. A typical ANN architecture is shown in Figure 5a. Apparently, ANN evaluates each feature element independently and, therefore, cannot process sequential features.

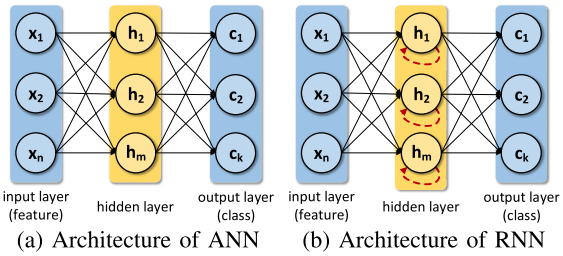


Fig. 5. ANN vs. RNN.

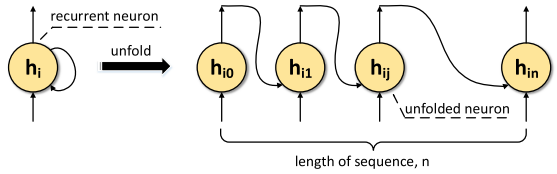


Fig. 6. An unfolded recurrent neuron in RNN.

RNN considers the sequential information through a simple modification of the traditional ANN. Specifically, in RNN, a self-feedback is applied on each neuron so that its outputs rely not only on inputs from the last layer but also on its own previous computations. For better understanding, an RNN can be converted into the traditional ANN through unfolding the feedback of its neurons, as shown in Figure 6, while the depth of the unfolded network depends on the length of the input sequence. Through these means, RNN *memorizes* information of what has been calculated and, therefore, leverages the sequential information in the input sequence. A typical architecture of an RNN is illustrated in Figure 5b.

2) Handling Sequential Features: Conventional ANN training, i.e. *backpropagation through time (BPTT)*, generally relies on the backpropagation of error and the gradient descent algorithm. However, the gradient-based training method may suffer from the *vanishing gradient* problem, as identified in [40]. In particular, traditional BPTT updates the weights backwards layer-by-layer by the chain rule, where the error at an arbitrary neuron is propagated back to its previous stages for multiple times, depending on the depth of the network. Correspondingly, the gradient decreases exponentially with the depth. When the depth is large, the gradient of the 'front' layers (i.e. layers closer to the initial inputs) may ultimately *vanish*. As a result, the BPTT algorithm may be undermined or not work at all. The RNN, unfortunately, is more severely affected by this problem, since its unfolded network structure is generally much deeper.

To overcome this limitation, we employ an alternative architecture of the RNN, namely *Long Short-Term Memory (LSTM)*, which was initially proposed in [41]. LSTM-RNN substitutes the original neuron with a *memory cell*, whose implementation is shown in Figure 7. A memory cell generally consists of an input gate, a neuron with self-feedback, a forget gate and an output gate [42]. The input gate determines whether the incoming signal can alter the current memory state or not while the output gate allows the outputs to affect other cells or blocks them. The forget gate, on the other hand, controls the effect of the previous memory state [41]. Through these means, LSTM maintains a more constant error propagation during BPTT training, which enables the RNN to learn over much longer steps, thereby preventing the vanishing gradient. In our implementation, we used the LSTM-RNN from Keras.

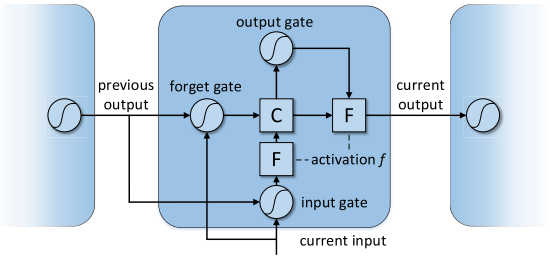


Fig. 7. The implementation of the memory cell in LSTM-RNN.

B. Outlier Detection

In order to identify unseen processes, which should not be classified as any existing process class, we rely on outlier detection. Specifically, two different approaches are considered, i.e., probability estimates and auto-encoder, to distinguish unseen process from seen process. The former leverages the independent features while the latter exploits the sequential features.

1) Probability Estimates: Our first outlier detection method makes use of 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 this 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.

2) Auto-Encoder: On the other hand, a more advanced outlier detection method is developed based on the auto-encoder. An auto-encoder is an ANN, which has exactly same dimensions in both the input layer and the output layer, and aims at learning the representative distribution of the inputs in order to reconstruct them at the outputs. The performance of an auto-encoder is generally evaluated through the reconstruction error, which indicates the deviation of the reproduced outputs from the inputs and can be implemented by the Mean Square Error (MSE). The reconstruction error, thus, is expected to be minimized in an optimized model. A typical auto-encoder is depicted in Figure 8.

In this work, in order to enable compatibility with the sequential inputs of the auto-encoder, we apply the same network structure as introduced in Section VI-A2, i.e., LSTM-RNN. Particularly, we attempt to learn the characteristics of a set of instruction sequences of a process through the auto-encoder so that the reconstruction error for elements in sequences of seen processes is minimized while the error for elements in sequences of unseen processes is distinguishably large. For each element i in a seen process sequence of length l, a maximum acceptable error $e_{max}(i)$ is learned in advance, hence, our outlier screening mechanism can be developed by setting a threshold on the value of abnormal reconstruction error as follows:

$$e_i = \begin{cases} e_{abr}, & if\ e_i > e_{max}(i) \\ e_{norm}, & \text{otherwise} \end{cases} \quad (1)$$

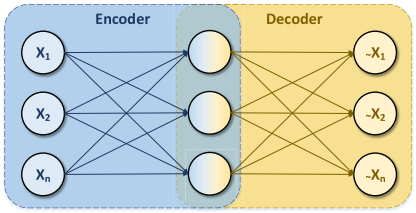


Fig. 8. Typical architecture of an auto-encoder.

$$sample = \begin{cases} seen\ proc., & if\ |\{e_i \mid e_i = e_{abr}\}| < th \\ outlier, & \text{otherwise} \end{cases} \quad (2)$$

Considering that the underlying distribution describing different seen process classes may vary significantly, in this work, we build a separate auto-encoder for each seen process classes and we individually set their corresponding thresholds.

VII. 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.

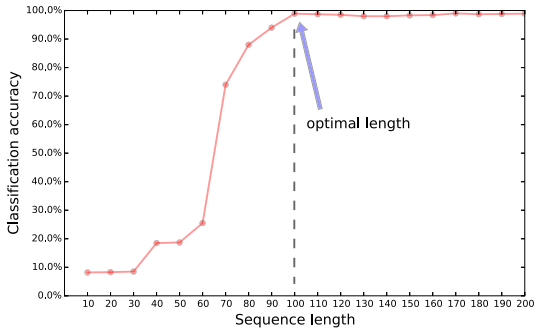
A. Experimental Setup

1) Setup for x86: Our experiments for x86 architecture were performed in Simics, wherein a 32-bit x86 machine was simulated with a single Intel Pentium 4 core running at 2 Ghz and containing 4 GB of RAM. A minimum installation Ubuntu server that embeds a Linux 3.8 kernel was then loaded as the OS on the simulated hardware. All collected data was normalized and fed to the analysis software via Python/MATLAB.

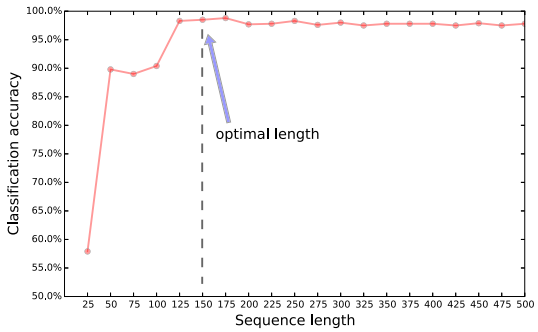
We use MiBench [43], a free commercially representative benchmark suite as our workload, which contains a few tens of application classes. The entire suite was executed 400 times, with each application invoked with various valid arguments or in the background (& option). The order of the workload execution was randomized to avoid the bias that a specific order might impose. We exploited the Simics feature, *haps*, to hook our event monitor on the iTLB and the program counter. In total, we collected a dataset containing approximately 9000 samples, where each sample represents a single process.

2) Setup for RISC-V: The experiments for RISC-V architecture, on the other hand, were performed in Spike, a RISC-V ISA simulator, where a 64-bit machine was simulated with 2 GB RAM. A basic RISC-V version of Linux 3.4 kernel was then loaded as the OS platform. Furthermore, the source code of the original Spike simulator was modified according to our specific purpose, i.e., monitoring instructions raising iTLB misses and changes of SPTBR value.

The MiBench benchmark was, once again, used as the workload. Unfortunately, 10 of the benchmark classes cannot be compiled by the RISC-V cross-compiler, due to the header file missing in the RISC-V Linux. Therefore, these testbenches were excluded in our experiments. Subsequently, the rest of the benchmark suite was executed 400 times in the same manner as in our previous experiments, resulting in approximately 4800 samples in total. Similarly to the previous case, each sample represents a single process.



(a)



(b)

Fig. 9. Classification accuracy over different input sequence lengths for x86 and RISC-V scenario. (a) x86 scenario. (b) RISC-V scenario.

B. Early Prediction Analysis Using LSTM-RNN

We first evaluate the early prediction effect in process reconstruction using LSTM-RNN approach on both x86 and RISC-V architectures. We note that LSTM-RNN accepts raw operator sequence as its input and performs multi-class classification on our workload dataset in order to reconstruct the executed processes. The dataset collected was split by half into the training set and the validation set. The RNN model consists of an embedding layer as its input layer, a fully-connected layer as its output layer, and an LSTM layer in between. The *sigmoid* function is used as the activation function for each layer. To train the model, a batch size of 64 and an epoch number of 100 were selected. The evaluated length of the operator sequence varied from 10 to 200 on x86 and from 25 to 500 on RISC-V. Figure 9 summarizes the performance of the LSTM-RNN classifier at different input sequence length on both architectures. As may be observed, the classification accuracy increases monotonically when the sequence length is increased, due to the fact that longer sequences convey more information for better distinguishability. On the other hand, input length of longer than 100 on x86 has no significant impact on the classification accuracy, which confirms the early prediction effect, indicating that process reconstruction based on the entire program execution flow is an overkill. Similarly, this phenomenon can also be observed on RISC-V after the input length reaches 150. As a result, in order to achieve acceptable classification accuracy with minimal logging overhead, we selected 100 and 150 as the optimal length of the input sequence on x86 and on RISC-V, respectively.

C. Effect of Extended Feature Set

To evaluate the effect of the extended feature set on different feature extraction strategies and various analysis methods,

TABLE III
PROCESS IDENTIFICATION RESULTS ON X86 (IN %)

Model / Benchmark	original feature set							extended feature set						
	CoO-KNN	CoO-SVM	3-gram KNN	3-gram SVM	4-gram KNN	4-gram SVM	RNN-LSTM	CoO-KNN	CoO-SVM	3-gram KNN	3-gram SVM	4-gram KNN	4-gram SVM	RNN-LSTM
toast	96.2	96.2	1.5	88.2	1.0	81.5	91.2	97.2	97.8	1.5	91.2	1.5	81.5	98.5
untoast	94.2	94.5	99.0	4.2	99.2	17.8	96.8	95.5	94.5	99.2	12.5	99.2	26.5	97.0
sha	100	100	96.5	92.5	93.8	98.5	96.5	100	100	97.2	93.8	96.5	98.8	100
lame	100	100	98.8	98.5	99.0	99.5	95.2	100	100	100	99.5	100	99.8	100
susan	100	100	98.8	98.8	97.8	99.0	98.5	100	100	99.5	99.5	97.8	99.5	100
basicmath	92.2	90.2	97.5	97.2	96.5	97.5	96.8	94.2	90.5	97.5	98.2	97.5	98.5	100
bf	97.5	98.0	98.2	97.0	98.0	97.5	99.5	100	99.5	99.5	98.8	99.5	97.8	100
patricia	98.0	98.2	78.5	94.5	98.5	98.0	99.2	99.5	98.5	78.2	94.5	98.8	98.2	99.5
search	97.8	97.0	100	99.5	99.0	99.0	99.0	99.0	98.2	100	99.5	99.2	99.8	99.0
fft	100	100	97.5	98.2	98.5	97.8	94.5	100	100	99.0	99.2	99.2	97.8	98.2
dijkstra	95.8	95.8	99.0	98.5	95.5	97.8	97.8	98.2	95.8	99.2	98.5	95.8	97.5	100
tiff2rgba	100	100	98.8	99.2	99.0	96.5	97.2	100	100	100	99.2	100	96.5	99.5
tiffmedian	96.2	100	65.2	76.5	74.2	89.2	90.2	96.8	100	68.5	81.5	74.5	90.5	98.5
tiff2bw	97.8	94.0	92.5	80.2	89.8	83.5	92.5	99.0	96.2	92.5	85.2	91.5	84.8	99.8
qsort	100	100	97.8	98.5	94.0	95.8	97.5	100	100	98.2	98.5	94.8	97.5	100
cjpeg	100	100	72.5	43.2	91.5	65.0	98.5	100	100	75.8	45.8	93.2	68.5	100
djpeg	97.0	100	48.2	45.2	66.8	75.8	84.5	97.5	100	50.2	47.8	49.2	78.2	100
rawcaudio	92.0	92.0	99.0	96.2	98.8	97.8	94.0	94.5	94.5	99.0	97.5	99.2	98.0	97.5
rawdaudio	51.8	52.2	61.8	56.8	58.5	37.0	95.2	56.0	58.5	62.5	57.5	58.0	42.2	96.5
crc	98.0	98.0	84.5	79.5	89.2	74.5	91.2	98.5	99.0	86.0	81.8	90.0	76.2	98.8
madplay	100	100	99.5	95.2	98.0	93.5	99.2	100	100	99.5	96.2	98.2	94.5	100
lout	100	100	86.5	91.2	93.8	95.8	95.5	100	100	88.2	93.0	95.0	96.5	100
pgp	97.5	98.0	43.2	72.0	70.5	88.0	89.5	99.2	99.5	47.5	76.2	74.0	90.2	97.0
overall	95.74	95.83	83.25	82.64	86.99	85.93	95.22	96.74	96.63	84.29	84.58	87.06	87.36	99.12

herein, a comprehensive horizontal analysis was performed. Table III and Table IV summarizes the comparison. In both tables, the left part contains data performed on the original feature set (mentioned in Section V-A), while the right part contains data performed on the extended feature set (mentioned in Table II). As may be observed, the extended feature set results in only similar overall process identification accuracy for Counts of Occurrence (CoO) model and n-gram model on both x86 and RISC-V architecture. On the other hand, the impact of the extension of the feature set is significantly observed on RNN model, i.e., about 4% increase in the overall accuracy on x86 and about 6% increase in the overall accuracy on RISC-V. The observation indicates that the RNN model with early prediction effect is more sensitive to feature resolution. This can be explained since the RNN model operates directly on the raw operator sequence without any processing and, thus, higher feature resolution carries more information for the RNN model to mine automatically. As a result, in the following sections, for the CoO model and the n-gram model, we mainly focus on the results achieved with the original feature set. For the RNN model, we mainly focus on the results achieved with extended feature set.

D. Process Classification Results

As mentioned in Section VII-A1, on x86, the process classification was performed on samples of 23 classes of processes, where each class had 400 samples. on RISC-V, the experiments were performed on samples of 12 process classes, where each class contained 400 samples. For both architectures, the corresponding dataset was split by half into training set and validation set, each of which contains half of the samples for every process class. Using these two datasets, we compared the effectiveness of the three types of features combined with three different machine learning models in identifying different processes, as introduced in Section V and Section VI. Results are summarized in Table III and Table IV.

1) Using Counts of Occurrence: We first evaluated the classification performance using the counts of occurrence features on the two architectures. The initial dimensionality of the collected feature vector matrices were as large as 83612 and was reduced to 200 after applying PCA. The reduced feature matrices were fed into the two classifiers, i.e., KNN and SVM. On x86, this leads to the process classification results as illustrated by the first two columns in Table III. As may be observed, both classifiers performed very well in correctly classifying the processes, reaching an overall classification accuracy of 95.74% and 95.83% respectively. For most classes, this accuracy was even higher. The similar result can be observed on RISC-V. As illustrated by the first two columns in Table IV, an overall accuracy of 94.5% and 92.1% can be reached, respectively.

However, on x86 architecture, 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`. Alternative features with more advanced machine learning algorithms may potentially address this limitation. The same observation does not apply to RISC-V architecture, simply because the two benchmark programs cannot be compiled on the architecture, and thus, were excluded in the evaluation.

2) Using n-Gram Model: Alternative features using the n-gram model were evaluated next. We experimented with the 3-gram and 4-gram models, which were applied on the operator sequence on the two architectures (whose lengths are determined based on the evaluation in VII-B) to extract features since the early prediction effect was revealed. As described in

TABLE IV
PROCESS IDENTIFICATION RESULTS ON RISC-V (IN %)

Model / Benchmark	original feature set							extended feature set						
	CoO-KNN	CoO-SVM	3-gram KNN	3-gram SVM	4-gram KNN	4-gram SVM	RNN-LSTM	CoO-KNN	CoO-SVM	3-gram KNN	3-gram SVM	4-gram KNN	4-gram SVM	RNN-LSTM
basicmath	93.5	88.2	92.0	91.5	93.8	90.5	89.8	93.8	90.5	91.8	93.0	95.2	90.5	96.5
bitcnts	98.0	98.2	95.0	95.2	95.5	94.5	92.0	98.0	98.8	95.8	95.5	95.8	96.0	99.0
qsort	88.5	100	94.2	90.5	94.0	88.5	93.5	89.2	100	96.0	92.0	94.5	90.0	97.5
susan	98.0	99.2	94.5	93.2	98.0	92.8	94.2	98.0	99.5	94.5	94.8	98.5	93.2	99.0
dijkstra	95.2	95.0	93.8	100	94.5	100	96.5	95.5	95.2	95.5	100	95.5	100	98.5
patricia	96.0	60.5	98.5	95.5	98.5	95.5	91.5	96.8	65.2	99.0	96.5	99.2	95.8	96.5
search	92.8	93.5	97.2	90.2	96.8	90.5	89.5	94.2	94.8	97.5	90.5	97.0	91.2	98.2
bf	95.8	96.0	94.0	93.0	96.5	92.2	91.5	96.5	96.8	94.8	95.2	97.5	92.2	97.0
crc	97.5	96.2	95.5	94.8	90.2	94.5	87.5	97.5	97.0	95.8	95.0	90.2	94.8	98.8
fft	94.8	94.8	98.2	97.0	99.2	95.5	95.8	94.5	95.0	98.2	96.8	99.0	95.8	100
toast	93.0	92.2	96.0	1.5	95.2	22.0	91.2	93.5	92.8	95.5	6.8	95.5	24.2	97.0
untoast	91.5	91.2	0.5	93.5	0.5	93.2	84.8	91.8	91.5	0.5	93.8	0.5	93.5	96.5
overall	94.54	92.08	87.45	86.48	87.66	87.45	91.48	94.94	93.09	87.91	87.49	88.20	88.10	97.88

Section V-A, on x86, where we consider 6 types of operators, the initial dimensionality of the feature matrix generated by the 3-gram and 4-gram model were 216 and 1296, respectively. Similarly, on RISC-V, where we consider 5 types of operators, the 3-gram and 4-gram model generated feature matrices whose initial dimensionality were 125 and 625, respectively. Compared to the use of Counts of Occurrence, the dimensions of the feature matrix generated using n-gram model were significantly smaller.

The matrices were then fed into KNN and SVM to perform process classification. As indicated by Table III, the overall classification accuracy for the two classifiers using 3-gram model were 83.25% and 82.64%, while the accuracy for the two classifiers using 4-gram model were 86.99% and 85.93% on x86. Similarly, as illustrated by Table IV, an overall classification accuracy of 87.46% and 86.33% for the two classifiers using 3-gram model can be achieved, while an overall accuracy of 87.66% and 87.45% for the two classifiers using 4-gram model can be achieved on RISC-V. As expected, the 4-gram model, which captures information in a finer-grain manner, performed better than the 3-gram model. However, compared with the result of using CoO, the overall performance of the n-gram model was not competitive. Neither was the issue of distinguishing between the process *rawcaudio* and *rawdaudio* resolved. This may be explained by the fact that the n-gram models, similar to the counts of occurrence, preserve the frequency and order information in a lossy way, sacrificing potentially helpful information from the operands. Moreover, the n-gram models were applied with the early prediction assumption, while the CoO models were applied on the entire program execution flow. The information carried by CoO model was certainly more comprehensive, while the n-gram models with early prediction cannot compensate for the information loss. Nevertheless, the n-gram model generated a much smaller feature matrix, which implies dramatically reduced storage/computation overhead.

3) Using Raw Operator Sequence: Finally, we evaluated the effectiveness of process classification using the raw operator sequence with the deep learning model. The dimensionality of the feature matrix in this scheme is equal to the optimal length of the operator sequence, i.e., 100 on x86 and 150 on RISC-V, due to the early prediction effect. As shown by the last columns in Table III and Table IV, the RNN-LSTM model performs the best in process identification, achieving an average classification accuracy of 99.12% on x86 and 97.8% on RISC-V, which is approximately 3% higher than the accuracy achieved through KNN/SVM with counts of occurrence. Furthermore, while the similarity issue between the process `rawdaudio` and `rawcaudio` was unresolved by using other features, the process `rawdaudio` was successfully distinguished from the process `rawcaudio` in this scheme. Indeed, essentially, counts of occurrence and n-gram model generates compressed representation of the raw instruction sequence. On the other hand, the raw operator sequence preserves the frequency and order information more precisely so the deep learning model can intelligently mine descriptive features from its input sequences. As a result, a lossless abstraction of the raw instruction sequence is generated, which leads to a better performance for process classification.

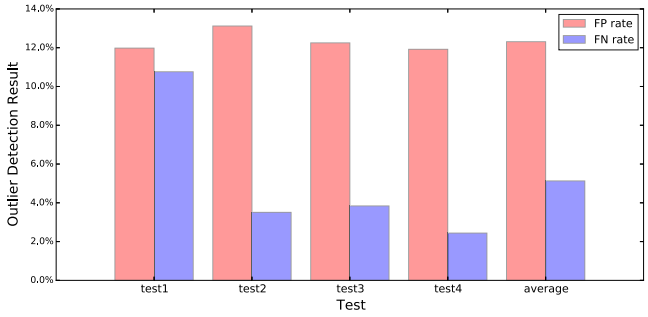


Fig. 10. Outlier detection results using probability estimates on x86.

E. Outlier Detection

To evaluate the effectiveness of the TPE in identifying unseen processes, we repeated the experiment, this time omitting 5 randomly selected classes from the training set, while retaining them in the validation set to mimic outliers. We compared the performance of our two outlier detection methods, i.e., probability estimates and auto-encoder, as follows.

1) Using Probability Estimates: We first evaluate the outlier detection using probability estimates. Through cross-validation, we set the threshold for outlier screening to 0.6, which is applied to the testing set to identify unseen process. Figure 10 and Figure 11 summarize the results for the repeated runs on x86 and RISC-V architectures. For each run, we report the false positive (FP) (i.e., seen process classified as outlier) and false negative (FN) (i.e., outlier classified as seen process)

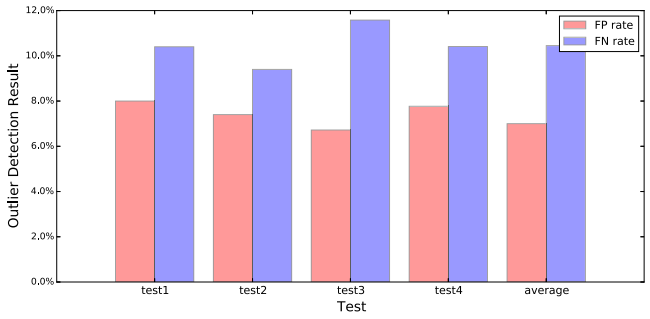


Fig. 11. Outlier detection results using probability estimates on RISC-V.

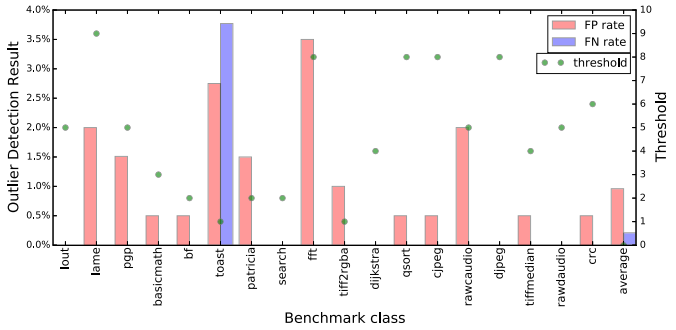


Fig. 12. x85 outlier detection results using auto-encoder.

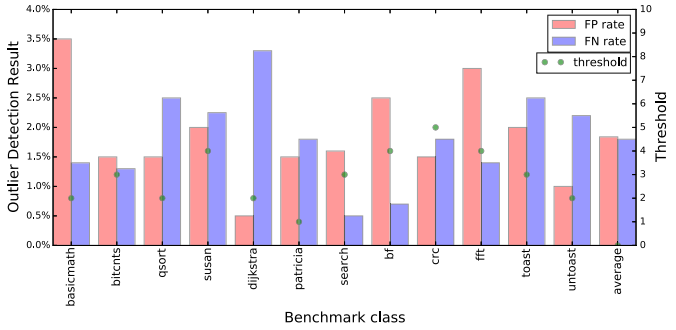


Fig. 13. RISC-V outlier detection results using auto-encoder.

rates. As may be observed, the simple outlier screening method described above results in high outlier detection accuracy, with the average FP and FN rate at 12.31% and 5.13% on x86, and the average FP and FN rate of 7% and 10.45% on RISC-V. Indeed, for previously seen processes, the probability difference between the top two classes is overwhelmingly high, while for outlier processes it is overwhelmingly low. Additionally, threshold adjustment can support biased decisions, favoring one error direction.

2) Using Auto-Encoder: Unlike probability estimates using a global threshold to identify outliers, an auto-encoder is built for each seen process class, which results in 18 independent auto-encoders on x86 architecture and 12 independent auto-encoders on RISC-V architecture respectively, whose thresholds to screen outliers were set separately. For each class, we report its corresponding threshold, as well as the FP/FN rate, which is summarized in Figure 12 and Figure 13. The left y axis represents the FP/FN rate while the right y axis represents the threshold applied for each process class to screen outliers.

As may be observed, this approach significantly reduces the FP/FN rate, compared with the method using probability estimates. On x86, it results in an average FP rate of 0.96% and an average FN rate of 0.21%. Zero FP or FN rate, which indicates no error in identifying outliers, can even be reached in certain process classes, while the worst case of the FN and FP rate is 3.5% and 3.77% respectively. Similarly, on RISC-V, an average FP rate of 1.84% as well as an average FN rate of 1.8% can be achieved. The worst case of the FN and FP rate is 3.3% and 3.5% respectively.

Indeed, modeling the characteristic distribution of different process classes individually may lead to a more precise interpretation of each class. Additionally, the sequential features also surpass the independent features in extracting meaningful information from the raw instruction sequence, which has been verified in the process classification results. Overall, outlier detection using auto-encoder can be expected to outperform alternatives using probability estimates.

F. Logging Overhead

To evaluate the design overhead of the TPE on both architectures, we focus on assessing the required data logging rate corresponding to our different feature extraction mechanisms. As discussed in Section V-A, the data pre-processing on the RISC-V differs from the x86 counterpart due to the distinct instruction set design. However, the methodologies for feature extraction are identical on both architectures and, thus, the same estimation method can be applied to both architectures. On the other hand, unfortunately, Simics is not a cycle-accurate simulator. Therefore, to attain a more accurate estimation, we calculated the logging rate as follows.

In the scheme of feature extraction using counts of occurrence, for each partition of a process, the TPE 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:

$$F.V.\ size = 18 \times \lceil \log_2 partition\ size \rceil \quad (3)$$

$$partition\ generation\ rate = \frac{iTLB\ miss\ rate}{partition\ size} \quad (4)$$

$$bits/inst. = F.V.\ size \times partition\ generation\ rate \quad (5)$$

$$est.\ logging\ rate(bits/sec) = \frac{bits/inst. \times clk\ freq.}{CPI(assumed = 1)} \quad (6)$$

On the other hand, in the scheme of feature extraction using n-gram model or raw operator sequence, it is more efficient to log the operator sequence itself directly. Given that the number of operator categories is 6 or 13, respectively, we only need 3 or 4 bits for each element in the sequence. Similarly, the number of categorized operators in the operator sequence to be logged per second is determined by the iTLB miss rate. The estimated logging rate is calculated by the equations below:

$$element\_size = \lceil \log_2 number\ of\ op.\ categories \rceil \quad (7)$$

$$bits/inst. = iTLB\ miss\ rate \times element\ size \quad (8)$$

$$est.\ logging\ rate(bits/sec) = \frac{bits/inst. \times clk\ \ freq.}{CPI(assumed = 1)} \quad (9)$$

We ran our benchmark suite several times to obtain an average iTLB miss rate. On x86, this value was 0.0016%, resulting in an estimated data logging rate of only 5.17 KB/s, 12.31 KB/s and 16.41 KB/s respectively. While a typical TLB miss rate is expected to be around 0.01-1% [44], 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 may be observed, compared with counts of occurrence, n-gram model and raw operator sequence nearly doubles/triples the logging rate. Nevertheless, due to the early prediction effect, the logging mechanisms in the latter two schemes are only enabled for the first 100 instructions raising iTLB miss while remaining disabled during the rest of the time, thereby generating logs of much smaller size. Therefore, while introducing higher rate during the logging process, n-gram model and raw operator sequence incur less storage overhead.

Correspondingly, on RISC-V, the average iTLB miss rate for user-space instructions was 0.026%. Moreover, the Spike simulator does not implement a timing model. Therefore, assuming the clock frequency of a prototyped RISC-V CPU, i.e., SiFive E51 which runs at 1.5 GHz [45], an average data logging rate of 71.7 KB/s, 146 KB/s and 195 KB/s is required when counts of occurrence, n-gram model and raw operator sequence are applied in feature extraction, respectively. The relatively high iTLB miss rate, as compared with the case of x86, can be explained by the different implementation of the RISC-V ISA as well as its cross-compiler. Further optimization on this platform, which is orthogonal to our work, is expected to lower the iTLB miss rate.

G. Summary

To recap, it has been revealed that the TPE is capable of identifying processes on both x86 and RISC-V architectures. Among all the feature extraction strategies and analysis models, the RNN model with extended feature sets yields the best identification accuracy. Nevertheless, compared with CoO model with traditional machine learning methods, the RNN model incurs a higher logging overhead. Since TPE follows an offline analysis model, we can afford implementation complexity in the analysis software. In this case, the advanced model may be the best choice to implement in the TPE. On the other hand, an online solution can potentially be explored based on the same methodology. While the CoO model with traditional machine learning algorithms yields slightly lower identification accuracy, it is much simpler to implement on-chip. In this situation, a more cost-effective solution may select the CoO model as the primary option to implement.

VIII. 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. We note that the hardware architecture of the proposed logging system is architecture-agnostic since the feature extraction methods are shared across both the architectures under evaluation.

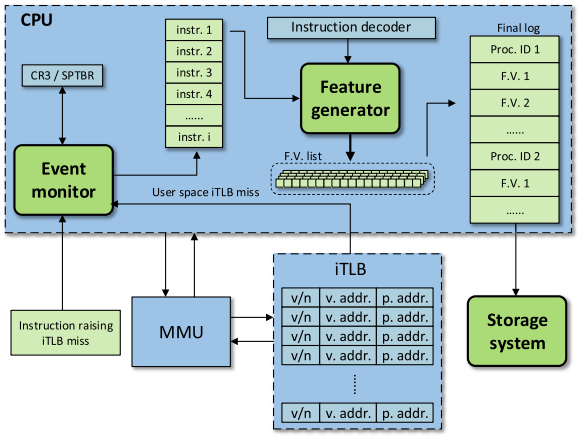


Fig. 14. Logging system implementation in hardware.

The hardware logging module consists of three main components, with its architecture shown in Figure 14:

Event Monitor: this component is used to monitor critical events, including TLB miss, CR3 register (x86) or SPTBR register (RISC-V) update, program counter update, etc. The event monitor serves as the main controller of the entire logging system. In modern architecture design, the TLB is implemented in the Memory Management Unit (MMU) and the 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 on x86 or the SPTBR on RISC-V, which works as an 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 V-D. When the length of the input instruction sequence reaches the optimal length (i.e., 100 for x86 and 150 for RISC-V), the feature generator notifies the storage system that the feature extraction for the current process has been accomplished.

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. 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.

Generally speaking, the innate immunity of hardware-based intrusion detection methods against software-based tampering comes at the cost of sacrificing flexibility. In particular, hardware-based methods performing online malware detection [37], [46], whenever the OS kernel is updated, require an updating of its underlying configuration. In order to maintain immunity, this information should not be modifiable through software or even the OS, hence, such updating is not at all straightforward. On the other hand, since the forensics analysis

in TPE is performed in software, our hardware implementation can remain unchanged during its lifetime, and thus, avoid this limitation.

IX. Discussion

A. Comparison With HPC-Based Solution

State-of-the-art Hardware Performance Counter (HPC)-based solutions, as summarized in [47], mainly focus on malware detection, which poses strict demands for online functionality. In contrast, TPE aims at workload forensics, which can be performed offline. Thereby, the TPE generally requires simpler on-chip logic than HPC-based solutions, since the analysis models can be implemented in software. Furthermore, unlike HPCs, which constantly monitor a wide range of system events, the TPE focuses exclusively on TLB miss events and corresponding instructions. Compared to HPC-based solutions, this feature model leads to lower logging bandwidth, while still retaining favorable process identification performance. As a point of reference, the HPC-based method in [46], which performs malware detection, requires bandwidth of a few hundred KB/s.

B. Environment Update

Hardware-based solutions, due to their nature, are not compatible to software environment updates such as OS kernel updates, workload updates, etc. Inevitably, after such updates, the analysis model may also require updates. Fortunately, the TPE follows an online-logging-offline-analyzing paradigm. Therefore, unlike other online, pure on-chip solutions, the required changes in the analysis model are relatively easy to implement. The hardware-related implementation in TPE involves only feature extraction and logging logic, which can, most likely, remain unchanged during environment changes.

X. Conclusion

We introduced TPE, a hardware-based framework for performing workload reconstruction for 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 collects the required information directly in hardware, making it impervious to such attacks. Herein, we demonstrated an incarnation of this general idea, i.e., TPE, which models the program behavior using machine learning algorithms based on instructions raising iTLB misses, in order to perform process identification as well as outlier detection.

The TPE was evaluated in Linux OS running on two representative architectures, i.e., 32-bit x86 and 64-bit RISC-V, which were simulated in the Simics and Spike simulators, respectively. Alongside the simulated hardware, a statistical analysis module was implemented, which employed KNN, SVM and RNN-LSTM for performing process classification, as well as probability estimates and auto-encoder for performing outlier detection. Comparison between their performance on both architectures indicates that the RNN-LSTM/auto-encoder model using the sequential features outperforms other analysis methods in terms of process classification accuracy/outlier detection accuracy as well as logging overhead. Specifically, experimental results using the popular Mibench benchmark suite reveal that, on x86, an overall process identification accuracy of 99.12% can be achieved, and an average FP/FN rate of 0.96% and 0.21% in identifying unseen process can be reached, at the cost of simple hardware additions capable of processing and logging data at a rate of 16.41 KB/s. Similarly, on RISC-V, an overall accuracy of 97.8% in process identification can be achieved, as well as an average FP/FN rate of 1.84% and 1.8% for unseen process identification can be reached, at the cost of data logging rate of 195 KB/s. These results corroborate the effectiveness as well as the generalizability of the TPE.

References

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

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

[3] J. Criswell, N. Dautenhahn, and V. Adve, "KCOFI: Complete control-flow integrity for commodity operating system kernels," in *Proc. IEEE Symp.*, Sep. 2014, pp. 292–307.

[4] T. Bletsch, X. Jiang, and V. Freeh, "Mitigating code-reuse attacks with control-flow locking," in *Proc. 27th Annu. Comput. Secur. Appl. Conf.*, 2011, pp. 353–362.

[5] C. Song, B. Lee, K. Lu, W. Harris, T. Kim, and W. Lee, "Enforcing kernel security invariants with data flow integrity," in *Proc. Netw. Distrib. Syst. Secur. Symp.(NDSS)*, 2016.

[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 *Proc. IEEE Symp.*, May 2012, pp. 586–600.

[7] S. Krishnan, K. Z. Snow, and F. Monrose, "Trail of bytes: New techniques for supporting data provenance and limiting privacy breaches," *IEEE Trans. Inf. Forensics Security*, vol. 7, no. 6, pp. 1876–1889, Dec. 2012.

[8] L. Litty, H. Lagar-Cavilla, and D. Lie, "Hypervisor support for identifying covertly executing binaries," in *Proc. 17th USENIX Secur. Symp.*, 2008, pp. 243–258.

[9] D. Perez-Botero, J. Szefer, and R. Lee, "Characterizing hypervisor vulnerabilities in cloud computing servers," in *Proc. Int. Workshop Secur. Cloud Comput.*, 2013, pp. 3–10.

[10] L. Zhou and Y. Makris, "Hardware-based workload forensics: Process reconstruction via TLB monitoring," in *Proc. IEEE Int. Symp. Hardw. Oriented Secur. Trust*, 2016, pp. 167–172.

[11] A. Mujumdar, G. Masiwal, and B. B. Meshram, "Analysis of signature-based and behavior-based anti-malware approaches," *Intl. J. Adv. Res. Comput. Eng. Technol.*, vol. 2, pp. 2037–2039, Jun. 2013.

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

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

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

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

[16] A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda, "AccessMiner: Using system-centric models for malware protection," in *Proc. 17th ACM Conf. Comput. Commun. Secur.*, 2010, pp. 399–412.

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

[18] J. Pfoh, C. Schneider, and C. Eckert, "Nitro: Hardware-based system call tracing for virtual machines," in *Proc. 6th Intl. Conf. Adv. Inf. Comput. Secur.*, 2011, pp. 96–112.

[19] N. Quynh and Y. Takefuji, "Towards a tamper-resistant kernel rootkit detector," in *Proc. ACM Symp. Appl. Comput.*, 2007, pp. 276–283.

[20] 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, Nov. 2003.

[21] X. Wang and R. Karri, "Reusing hardware performance counters to detect and identify kernel control-flow modifying rootkits," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 35, no. 3, pp. 485–498, Mar. 2016.

[22] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Transparent rop exploit mitigation using indirect branch tracing," in *Proc. 22nd USENIX Secur. Symp.*, 2013, pp. 447–462.
[23] L. Davi, P. Koeberl, and A.-R. Sadeghi, "Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation," in *Proc. 51st ACM/EDAC/IEEE Design Autom. Conf. (DAC)*, Jun. 2014, pp. 1–6.
[24] A. Kanuparthi, J. Rajendran, and R. Karri, "Controlling your control flow graph," in *Proc. IEEE Symp. Hardw. Oriented Secur. Trust*, May 2016, pp. 43–48.
[25] Y. Xia, Y. Liu, H. Chen, and B. Zang, "CFIMon: Detecting violation of control flow integrity using performance counters," in *Proc. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Jun. 2012, pp. 1–12.
[26] S. Das, Y. Liu, W. Zhang, and M. Chandramohan, "Semantics-based online malware detection: Towards efficient real-time protection against malware," *IEEE Trans. Inf. Forensics Security*, vol. 11, no. 2, pp. 289–302, Feb. 2016.
[27] A. Nazari, N. Sehatbakhsh, M. Alam, A. Zajic, and M. Prvulovic, "EDDIE: EM-based detection of deviations in program execution," in *Proc. 44th Annu. Int. Symp. Comput. Archit.*, Jun. 2017, pp. 333–346.
[28] P. Krishnamurthy, R. Karri, and F. Khorrami, "Anomaly detection in real-time multi-threaded processes using hardware performance counters," *IEEE Trans. Inf. Forensics Security*, vol. 15, pp. 666–680, 2020.
[29] S. Banin and G. O. Dyrkolbotn, "Multinomial malware classification via low-level features," *Digit. Invest.*, vol. 26, pp. S107–S117, Jul. 2018.
[30] L. Zhou and Y. Makris, "Hardware-assisted rootkit detection via on-line statistical fingerprinting of process execution," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2018, pp. 1580–1585.
[31] H. Sayadi *et al.*, "2SMaRT: A two-stage machine learning-based approach for run-time specialized hardware-assisted malware detection," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2019, pp. 728–733.
[32] L. Zhou, Y. Hu, and Y. Makris, "A hardware-based architecture-neutral framework for real-time iot workload forensics," *IEEE Trans. Comput.*, vol. 69, no. 11, pp. 1668–1680, 2020.
[33] Y. Zhang, L. Zhou, and Y. Makris, "Hardware-based real-time workload forensics via frame-level TLB profiling," in *Proc. IEEE 37th VLSI Test Symp. (VTS)*, Apr. 2019, pp. 1–6.
[34] Y. Zhang, L. Zhou, and Y. Makris, "Hardware-based real-time workload forensics," *IEEE Des. Test. Comput.*, vol. 37, no. 4, pp. 52–58, Aug. 2020.
[35] A. P. Kuruvila, S. Kundu, and K. Basu, "Analyzing the efficiency of machine learning classifiers in hardware-based malware detectors," in *Proc. IEEE Comput. Soc. Annu. Symp. VLSI (ISVLSI)*, Jul. 2020, pp. 452–457.
[36] L. Zhou and Y. Makris, "Hardware-based on-line intrusion detection via system call routine fingerprinting," in *Proc. DATE*, 2017, pp. 1546–1551.
[37] M. Ozsoy, C. Donovick, I. Gorelik, N. Abu-Ghazaleh, and D. Ponomarev, "Malware-aware processors: A framework for efficient online malware detection," in *Proc. IEEE 21st Intl. Symp. High Perform. Comput. Archit.*, 2015, pp. 651–661.
[38] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanović, "The RISC-V instruction set manual, volume I: User-level ISA, version 2.1," Dept. Elect. Eng. Comput. Sci., Univ. California Berkeley, Tech. Rep. UCB/EECS-2016-118, 2016. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-118.html
[39] C.-C. Chang and C.-J. Lin, "LIBSVM: A library for support vector machines," *ACM Trans. Intell. Syst. Technol.*, vol. 2, no. 3, pp. 1–27, Apr. 2011.
[40] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *IEEE Trans. Neural Netw.*, vol. 5, no. 2, pp. 157–166, Mar. 1994.
[41] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, 1997.
[42] F. A. Gers, J. Schmidhuber, and F. Cummins, "Learning to forget: Continual prediction with LSTM," *Neural Comput.*, vol. 12, no. 10, pp. 2451–2471, Oct. 2000.
[43] 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 *Proc. IEEE Int. Workshop Workload Characterization*, Sep. 2001, pp. 3–14.
[44] D. A. Patterson and J. L. Hennessy, *Computer Organization And Design Hardware/Software Interfac*. Burlington, MA, USA: Morgan Kaufmann, 2009.
[45] *E51 Risc-V Core Ip*. [Online]. Available: https://old-www.sifive.com/products/risc-v-core-ip/e5/e51/
[46] J. Demme *et al.*, "On the feasibility of online malware detection with performance counters," *ACM SIGARCH Comput. Archit. News*, vol. 41, no. 3, pp. 559–570, Jun. 2013.
[47] S. Das, J. Werner, M. Antonakakis, M. Polychronakis, and F. Monrose, "Sok: The challenges, pitfalls, and perils of using hardware performance counters for security," in *Proc. IEEE Symp.*, Oct. 2019, pp. 20–38.

Liwei Zhou (Member, IEEE) received the bachelor's degree in electrical engineering from Tongji University in 2007, the M.S. degree in electrical engineering from The University of Texas at Dallas in 2013, and the Ph.D. degree from the Department of the Electrical and Computer Engineering, The University of Texas at Dallas, in 2018. After his graduation, he joined ASML-HMI as a Software Engineer. He currently works as a Software Engineer in cloud networking domain with Google. His research interest includes trustworthy security-enforced computer architecture for system security applications, e.g., computer forensics and malware detection.

Yunjie Zhang (Graduate Student Member, IEEE) is currently pursuing the Ph.D. degree in electrical and computer engineering with The University of Texas at Dallas, Richardson, TX. His research interests include application of machine learning in workload forensics and malware detection.

Yiorgos Makris (Senior Member, IEEE) received the Diploma degree in computer engineering from the University of Patras, Greece, in 1995, and the M.S. and Ph.D. degrees in computer engineering from the University of California at San Diego, San Diego, in 1998 and 2001, respectively. After spending a decade on the faculty of Yale University, he joined UT Dallas, where he is currently a Professor of electrical and computer engineering, leading the Trusted and RELiable Architectures (TRELA) Research Laboratory, and the Safety, Security and Healthcare Thrust Leader for Texas Analog Center of Excellence (TxACE). His research interests include applications of machine learning and statistical analysis in the development of trusted and reliable integrated circuits and systems, with particular emphasis on the analog/RF domain. He was a recipient of the 2006 Sheffield Distinguished Teaching Award, Best Paper Awards from the 2013 IEEE/ACM Design Automation and Test in Europe (DATE 2013) Conference and the 2015 IEEE VLSI Test Symposium (VTS 2015), as well as Best Hardware Demonstration Awards from the 2016 and the 2018 IEEE Hardware-Oriented Security and Trust Symposia (HOST 2016 and HOST 2018). He has served as an Associate Editor for the IEEE TRANSACTIONS ON INFORMATION FORENSICS AND SECURITY and the *IEEE Design and Test of Computers* Periodical, and a Guest Editor for the IEEE TRANSACTIONS ON COMPUTERS and the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS. He serves as an Associate Editor for the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS.