HARDWARE-BASED WORKLOAD FORENSICS

AND MALWARE DETECTION IN MODERN MICROPROCESSORS

by

Liwei Zhou

APPROVED BY SUPERVISORY COMMITTEE:

Yiorgos Makris, Chair

Carl Sechen

Benjamin Carrion Schaefer

Yang Hu

This dissertation

is dedicated to my dear, departed grandparents and

to my beloved grandmother-in-law.

I miss you so much.

HARDWARE-BASED WORKLOAD FORENSICS AND MALWARE DETECTION IN MODERN MICROPROCESSORS

by

LIWEI ZHOU, BS, MS

DISSERTATION

Presented to the Faculty of

The University of Texas at Dallas

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY IN

ELECTRICAL ENGINEERING

THE UNIVERSITY OF TEXAS AT DALLAS

December 2018

ACKNOWLEDGMENTS

I would like to acknowledge my advisor, Dr. Yiorgos Makris, for the continuous support of my PhD study and related research, for his patience and motivation. His empathy encouraged me when I fell into frustration, and his guidance helped me in understanding how to explore a research problem from scratch and improving my writing skills and presenting academic papers. Thank you for helping me grow and become confident and mature.

I am very thankful to my labmates. Most of us joined the group at the same time. We shared ideas. We complained of the obstacles and frustration, but we encouraged each other to survive. We worked together to achieve several awards for this lab. Thank you for making this group a family.

Last but not least, I would like to thank my family: my parents and my girlfriend, for supporting me spiritually throughout my PhD life. I feel sorrow for the times when I could not be with you, and I appreciate your patience, tolerance, and understanding. I love you all.

November 2018

HARDWARE-BASED WORKLOAD FORENSICS

AND MALWARE DETECTION IN MODERN MICROPROCESSORS

Liwei Zhou, PhD
The University of Texas at Dallas, 2018

Supervising Professor: Yiorgos Makris, Chair

Traditional computer forensics and/or malware detection methods are generally implemented at the operating system (OS) or the hypervisor level, which benefits from abundant software semantics and implementation flexibility. Nevertheless, the data logging and monitoring systems involved in these methods are vulnerable to spoofing attacks at the same level, which undermine their effectiveness. In this dissertation, the hardware-based methodologies are proposed to perform workload forensics and/or malware detection in microprocessors. In contrast to the software-based counterparts, a hardware-based implementation ensures the immunity to software tampering. Specifically, a generic architecture is introduced which a hardware-based forensic analysis or a malware detection method needs to follow, as well as the various architecture-level information which could potentially be harnessed to ensure system security and/or integrity. To illustrate the proposed concept, two incarnations, i.e., hardware-based workload forensics and hardware-based rootkit detection are present. Experimental results corroborate that even a low-cost hardware implementation can facilitate highly successful forensics analysis and/or malware detection, while taking advantage of its innate immunity to software-based attacks.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 Motivation

Over the last few decades, the prevalence of electronic devices has resulted in rapidly increasing amounts of private/sensitive information, such as personal details or trade secrets, being stored, processed and exchanged in electronic form. Unfortunately but inevitably, this has also lead to the emergence of hundreds of millions of malicious software [32], or *malware*, which seek to interfere with the underlying computer systems and to steal or disrupt such information, in order to benefit from such illegitimate access. As a result, developing defense mechanisms against these threats becomes indispensable. Generally speaking, implementation of such defense mechanisms can branch into computer forensics and malware detection. The former aims at performing retroactive investigations to reconstruct past events while the latter aims at detecting and/or preventing the execution of potential threats.

Most state-of-the-art computer forensics and malware detection methods are software-based, i.e., developed at OS-level or hypervisor-level. OS-level methods benefit from semantic-rich information, e.g., process ID, file system objects, etc., as well as flexible deployment. Nevertheless, they are susceptible to software attacks launched from the same privilege domain. To address this limitation, hypervisor-level methods were proposed, since hypervisors operate with higher privileges. Unfortunately, the hypervisor itself can be the attack target, as several vulnerabilities and intrusion methods have been identified [39]. Consequently, software-based detection approaches may suffer the risk of corruption of the logged data or even disabling of the detection system.

1.2 Contribution

To address the aforementioned limitation, in this dissertation, a hardware-based framework is proposed to perform computer forensics and malware detection. Specifically, this framework relies exclusively on data collected directly through the hardware, without the intervention of a hypervisor or an OS, whereby the logged information may be compromised. Accordingly, traces obtained from hardware are expected to be immune to software-based tampering. On the other hand, a hardware-based solution requires circuitry addition and modification in the microprocessor for identifying, extracting, and logging the relevant information. Therefore, judicious selection of information sufficient for fulfilling the targeted task becomes crucial. To this end, the proposed framework leverages architecture-level events, which are related to program execution in the OS, in order to perform the hardware-based defense with low cost. In this dissertation, I present several applications in both computer forensics and malware detection with various implementations, to illustrate the proposed idea.

1.3 Dissertation Organization

The remainder of the dissertation is structured as follows. Chapter 2 briefly discusses the related work. A generic architecture of the proposed hardware-based methodology as well as its potential design challenges are introduced in Chapter 3. Two incarnations, which perform hardware-based workload forensics and hardware-based rootkit detection, are briefly present to illustrate the proposed concept in Chapter 4 and Chapter 5, respectively. Chapter 6 summarizes the research and discusses the future work.

CHAPTER 2

LITERATURE REVIEW

The state-of-the-art in forensic analysis and malware detection methods are discussed in this chapter, which can be categorized into OS-level approaches and hypervisor-level approaches. Within each category, existing methods can be further divided into data-centric and program-centric, depending on the methodology employed in these methods.

2.1 OS-level approaches

OS-level approaches generally benefit from the semantic-rich information. Data-centric approaches in this category mainly focus on performing signature-based analysis to verify the integrity of objects of interest for forensic analysis or malware detection. Various commercial computer forensic products fall into dis paradigm. For example, EnCase creates images for disk data to enable data recovery and/or to ensure data integrity. Similar products include FTK and Registry Recon [22, 2, 5]. On the other hand, detecting malware through Control Flow Integrity (CFI), which seeks to identify illegitimate redirection of program control flow, has been proposed as a promising defense against control flow hijacking attacks of OS kernel services [15, 7]. Alternatively, program-centric approaches model the program behavior based on information related to the program execution flow, e.g., system call sequence, to perform further analysis. A large body of work on intrusion detection follows this paradigm [30, 46, 9]. In general, these methods rely solely on analysis of system call sequences. An interesting extension is introduced in [35], 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, 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 [33].

2.2 Hypervisor-level approaches

Hypervisor-level approaches benefit from the inherently higher security offered by virtualization and isolation, as mentioned in Chapter 1. Nevertheless, these approaches 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 uses the CR3 register available in the x86 architecture in order to identify process creation, switching and termination [26]. Once the semantic gap is bridged, program-centric intrusion detection 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 [21, 40]. 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 [36]. Such methods even go to a lower layer, to check whether contents on the disk and its image in main memory match [31, 34]. Nevertheless, they still rely on OS-level information (e.g., system.map) to locate which part is critical to keep their eyes on [34].

CHAPTER 3

HARDWARE-BASED DEFENSE METHODOLOGY

3.1 System Design

The generic design of the proposed hardware-based methodology for forensics analysis or malware detection, as shown in Figure 3.2, involves three dimensions: (1) designing the system architecture of the approaches, which determines how data should be collected, processed, and analyzed, (2) selecting objects of interest according to the specific objective, and (3) selecting appropriate analysis methods to process the collected data for the corresponding purpose. More specifically, we evaluate several options corresponding to different application scenarios in each dimension, which will be discussed in detail in following sections.

3.1.1 System Architecture

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

The data analysis component, on the other hand, is responsible for performing specific analyses on the logged data, while its actual implementation depends upon the type of defense mechanism to be employed. Unlike the data logging component, the analysis component can be performed either in the hardware, to enable on-line functionalities for prompt response, or in a trusted software environment, to perform off-line analysis with flexibility. Figure 3.1 illustrates the generic system architecture.

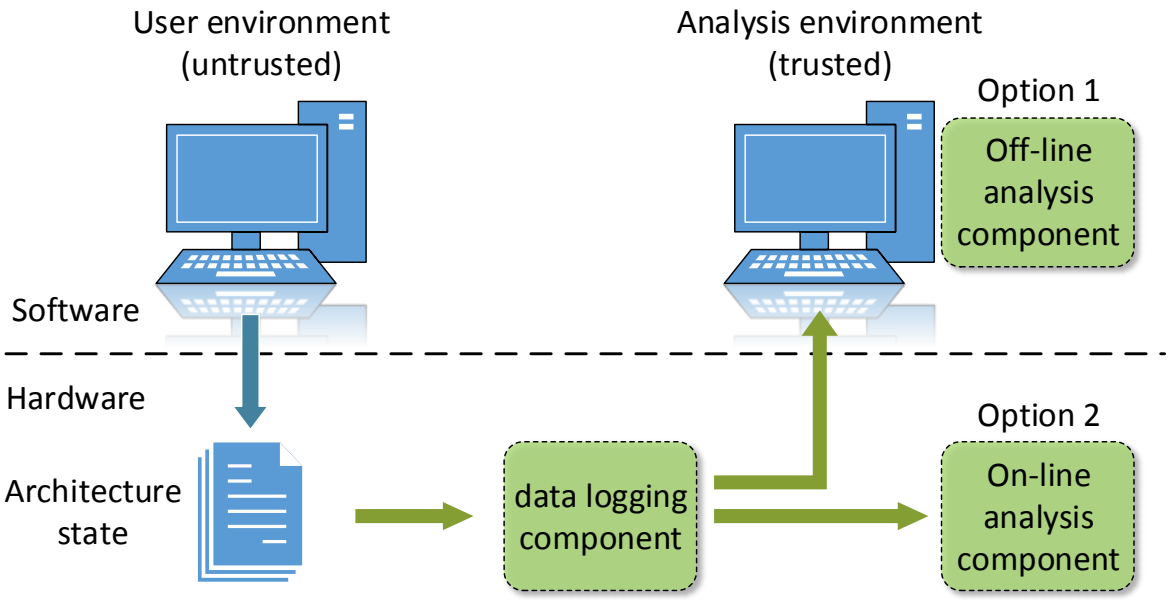


Figure 3.1: High-level system architecture

3.1.2 Object of Interest

Depending on the objective, a defense methods can be categorized into *data-centric* or *program-centric*. Data-centric methods generally focus on the integrity of a piece of specific data in order to investigate whether any unauthorized modification occurs. Given the nature of malicious software, popular objects of interest in this category include kernel image, kernel service table, control flow of kernel service, network channel, etc. This option is generally employed in the scenarios that invariants (e.g., rule of execution, static code behavior, network protocol, etc.) must be maintained and a *complete* set of the golden references is available. Moreover, It tends to benefit from a light implementation overhead due to the simple comparison.

Program-centric methods, on the other hand, model the expected behavior of a program in order to identify what program it actually is or whether it is malicious. The OS-level abstraction of a program, i.e., *process* and its dynamic execution flow, are the common objects

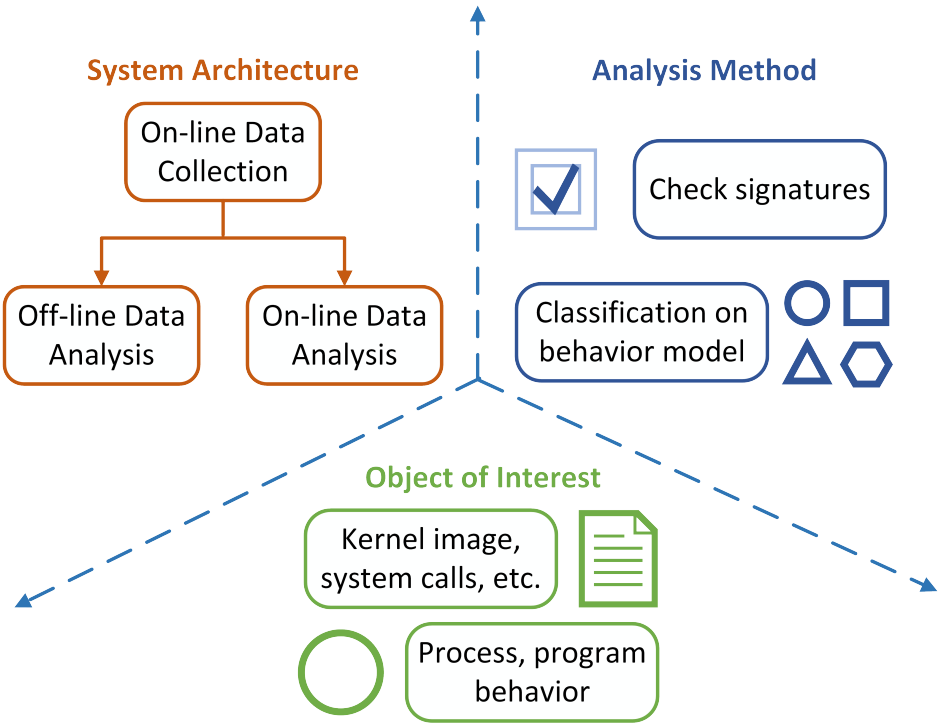


Figure 3.2: Three dimensions of the design of the proposed methodology

of interest in this category. This option is preferred in the more complicated scenarios that a complete set of golden reference is unachievable and thus, a straightforward comparison cannot be applied. For example, the complete set of program control flow is deemed to be unpredictable statically since some of the program execution path depends on its dynamic input arguments. Accordingly, a statistical model needs to be involved in order to evaluate the program behavior and draw conclusion with certain level of confidence (e.g., in terms of probability). Nevertheless, this option, compared to its former counterpart, definitely incurs more implementation overhead.

3.1.3 Analysis Methods

Similarly, the analysis methods employed herein fall into either *signature-based* methods or *behavior-based* methods. Signature-based methods generate *checksums* over their objects of

interest, which can be used as a golden reference for integrity checking or as a description of the expected behaviors. These methods benefit from their simplicity of implementation and may work well with those objects whose execution is fixed or infrequently changed. Given the complexity of program execution, however, behavior-based methods are more favorable when the dynamic behavior of a program has to be learned. These methods aim at modeling program behavior dynamically based on a number of pre-defined 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 algorithms and statistical analysis are typically employed.

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

CHAPTER 4

HARDWARE-BASED WORKLOAD FORENSICS[1,2]

In this chapter, the feasibility of workload forensics using hardware-based methodologies is evaluated. Specifically, two incarnations of off-line workload forensics based on spatial features and temporal features are introduced, which involves hardware-software co-design while an on-line workload forensics solution is proposed, which potentially benefits the application scenario of identifying workloads in real time.

4.1 Off-line Workload Forensics using Spatial Features

4.1.1 System Overview

In this work, we experiment with one instantiation of hardware-based workload forensics, whose actual implementation follows an off-line system architecture, i.e., performing the data logging in hardware while analyzing collected data in software. Specifically, we explore the possibility of reconstructing workload at the granularity of a process, while relying solely on information available through monitoring the TLB and statistically processing this information. Considering our objective of developing a hardware-based solution, however, we need to address the semantic gap problem. Indeed, we need to identify a process directly at the circuit level (i.e., without relying on data available at the OS level), so that we can associate with it the logged information that will be used for workload reconstruction. In modern OSs, due to the virtual memory concept, each process has its own dedicated address space, which maps resources used by the process into physical memory. This mapping is facilitated by the

[1] © 2016 IEEE. Reprinted/portions adapted, with permission, from Liwei Zhou and Yiorgos Makris, "Hardware-based workload forensics: Process reconstruction via TLB monitoring," in IEEE International Symposium on Hardware Oriented Security and Trust (HOST), May 2016, pp. 167-172.

[2] © 2016 IEEE. Reprinted/portions adapted, with permission, from Liwei Zhou and Yiorgos Makris, "Hardware-based workload forensics and malware detection in microprocessors," in 17th International Workshop on Microprocessor and SOC Test and Verification (MTV), December 2016, pp. 45-50.

translation between virtual address and physical address, maintained by a per-process page table. In x86, the base address of this table is stored in a control register, `CR3`. Changes of the `CR3` value perfectly match the events of process creation, switching and termination [26]. Accordingly, by monitoring the CR3 register, delineating processes becomes possible, thereby bridging the semantic gap. Below, we provide details of the two key components of our system, namely the logging module and the analysis module.

4.1.2 Implementation

Logging Module

Program execution typically follows phases, which can be effectively predicted via performance counter values [19]. Performance counters, however, generally contain global values, reflecting performance of a microprocessor over its entire workload. Moreover, order of program execution will affect performance counter values. As a result, bridging the semantic gap and associating these values accurately with OS-level objects, such as processes, is not at all straightforward.

To address this limitation, rather than using performance counter values, our approach uses *instructions causing TLB misses* as its main logging object. A TLB is a small cache memory which maintains recent translations of virtual addresses to physical addresses. In x86, when the CR3 value changes, the entire TLB is flushed. This design convention benefits our approach in two ways. First, all TLB events can be accurately associated with the process represented by the current CR3 value. Second, the effect of different order of program execution is mitigated, as the TLB starts fresh with every process. Therefore, the granularity of the logged data (i.e., process-level) matches our analysis target.

In x86, the TLB is split into two parts, one for instruction addresses (iTLB) and the other for data access addresses (dTLB). The logging module monitors the iTLB state and identifies the instructions which raise an iTLB miss. Only user-space instructions are considered in

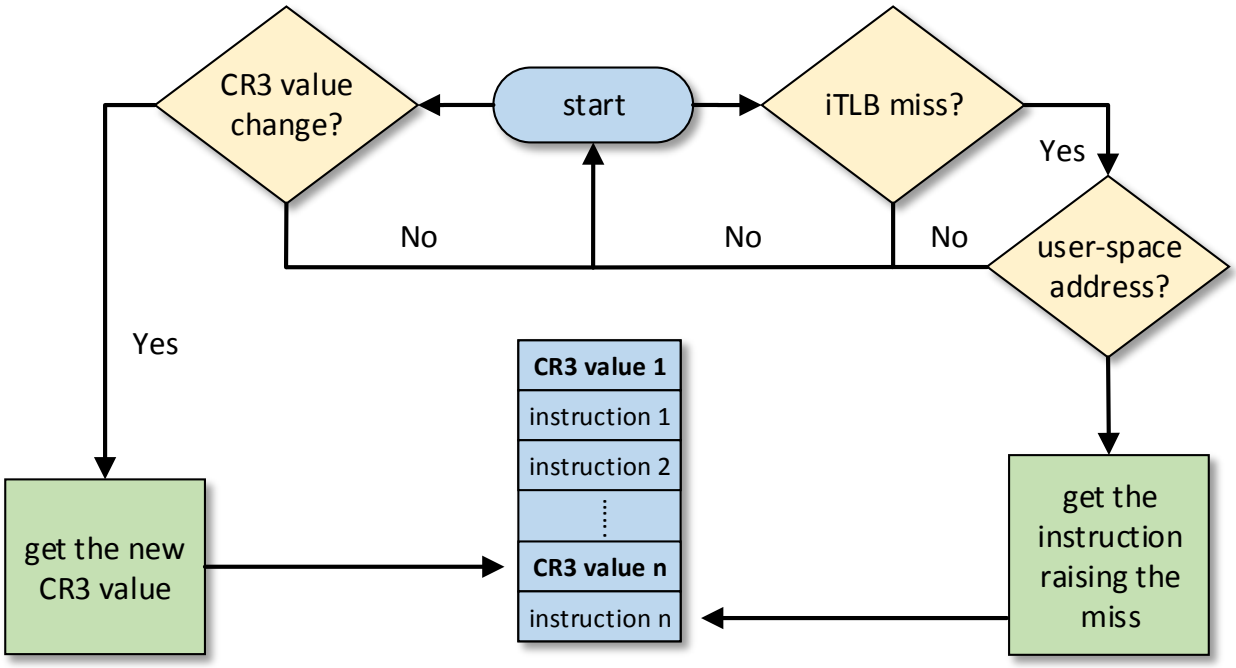


Figure 4.1: Logging logic

our scheme. In the Linux OS, all virtual addresses higher than `0xC0000000` are regarded as pointers to kernel space. Accordingly, our logging module ignores iTLB miss events raised by such addresses. In the end, each CR3 value, which represents a separate process, can be associated with a sequence of instructions (which caused iTLB misses). Figure 4.1 shows the logging logic.

In order to use machine learning for analysis, we extract a normalized set of features from the logged data. In our scheme, we use features which reflect both order and frequency information. Conceptually, for each CR3 value, its associated set of instructions causing iTLB misses is first partitioned into subsets of a maximum size of `partition_size`. Partitioning helps retain order information while reducing log size. In one extreme, choosing `partition_size` to be 1 retains all instruction order information but is too expensive and, most likely, unnecessary. In the other extreme, no partitioning would minimize the log size but would also sacrifice all order information, thereby limiting the accuracy of the foren-

sic analysis. In our system, we experimented with `partition_size` of 100 instructions. In practice, to minimize the required hardware, we do not log the actual instructions in each partition but, rather, a set of 18 frequency features. These 18 features are extracted through counters which are updated every time a qualifying iTLB miss occurs, and reflect information regarding the *operator* and the *operands* of the qualifying instruction, as shown in Figure 4.2.

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

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

The remaining twelve features capture the count of qualifying instructions which use the various types of operands (Opr.). These include 8 features corresponding to the 8 general purpose registers of 32-bit x86, one for memory reference, one for XMM registers and floating point stack, one for all segment registers, and one for immediate value.

A vector $F.V._i =< Op._1, ..., Op._6, Opr._1, ..., Opr._{12} >$ is extracted for each partition. For each process, as identified through its CR3 value, a list of feature vectors $[F.V._1, ..., F.V._i, ..., F.V._{end}]$ is collected, reflecting the order of partitions. The length of this list is considered as an additional feature. Ultimately, a feature matrix is generated, as shown in Figure 4.2. We

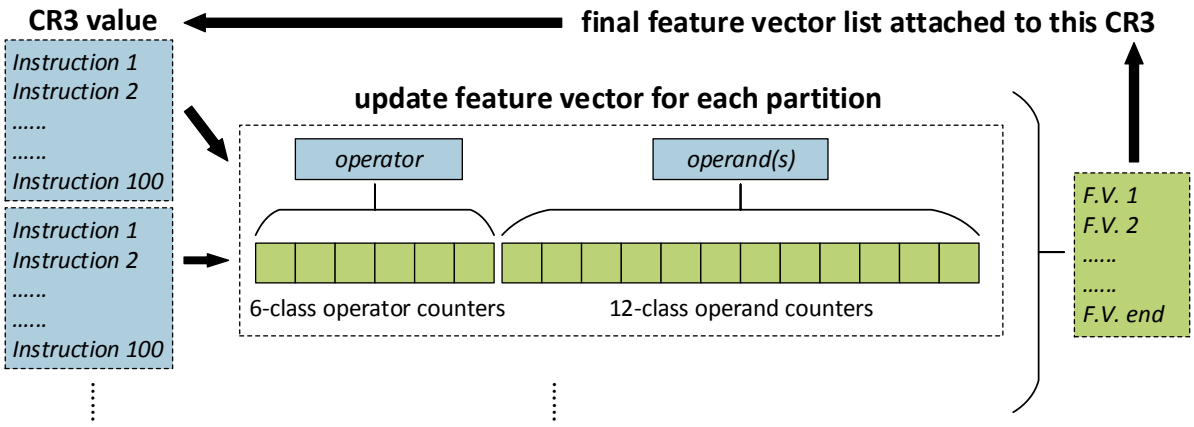


Figure 4.2: Feature extraction mechanism

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.

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

The hardware logging module consists of three main components, with its overall architecture shown in Figure 4.3:

Event Monitor: this component is used to monitor critical events, including TLB miss, CR3 register update, program counter update, etc. The event monitor serves as the main controller of the entire logging system. In x86, the TLB is implemented in the Memory Management Unit (MMU) and miss events are handled transparently by the hardware. The event monitor is expected to reside in the CPU but is also connected to the iTLB cache memory to get notification when a miss occurs. After the hardware resolves this miss (and

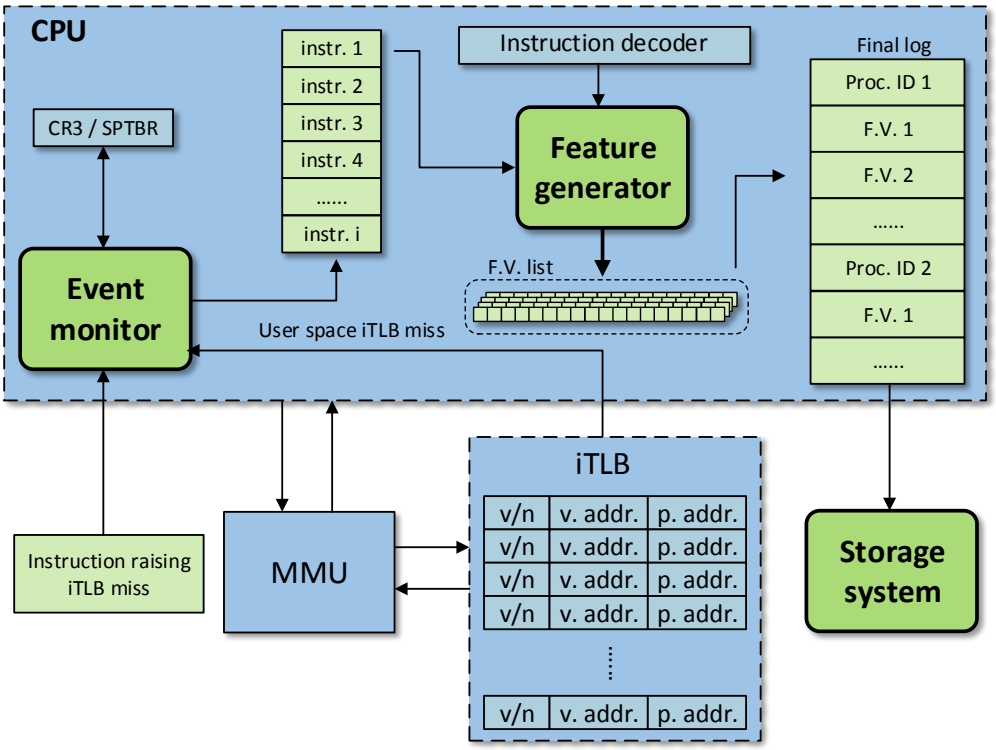


Figure 4.3: Logging system implementation in hardware

independently of whether a translation is found in the page table or not), the event monitor picks up the instruction which raised the iTLB miss and feeds it to the feature generator. In parallel, the value of the CR3 register, which works as an 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 above. A temporary register is used to update the values of a feature vector. When the partition size limit is reached or the current process terminates, the final value is sent to the storage system along with the CR3 value.

Storage System: this component is the actual space where the logged information is stored. A FIFO buffer is used to handle the clock difference between the CPU and the storage

system. To save memory space, zero padding is not done in hardware. Instead, the size discrepancy between log entries is handled during analysis. Periodically or continuously the logged data is transmitted through a dedicated port, which is physically inaccessible by the OS, to a trusted external storage or to the environment where analysis is performed.

Analysis module

The objective of the analysis module is to reconstruct workload execution at the granularity of a process, using the extracted feature matrices. Since forensics is typically an *ex post facto* effort, analysis is implemented in software and is executed in a trusted environment. However, future extensions could use dedicated on-chip learning to perform the analysis directly in hardware, possibly even in real-time, in a fashion similar to the malware detection method described in [19].

The actual analysis is based on machine learning and employs multi-class classification, where each class corresponds to a single process. Additionally, previously unseen processes are identified through outlier detection. We experimented with two different non-linear multi-class classifiers of varying complexity and performance, namely K-Nearest Neighbors (KNN) and Support Vector Machine (SVM). KNN computes the k nearest neighbors for a sample based on their Euclidean distance and assigns the sample to a class based on majority voting among these neighbors. SVM, on the other hand, generates decision boundaries which separate the feature space into labeled sub-spaces, while ensuring maximal separation among them. When evaluating a new sample, the SVM classifies it based on the label of the sub-space that it falls into. An important consideration when applying machine learning is the high dimensionality of the feature matrix. Since the extracted feature vector list may contain a large number of elements, it is necessary to reduce the dimensionality before performing classification, in order to avoid the curse of dimensionality. To this end, we use Principal Component Analysis (PCA), which generates a lower-dimensional feature matrix,

while retaining most of the information of the original matrix. In our implementation, we used KNN from the Matlab library and SVM from the LIBSVM library [11].

Upon the generated features, this method employs multi-class classification for workload reconstruction, where each class corresponds to a single process. Additionally, previously unseen processes are identified through outlier detection. Regarding process classification, two different non-linear multi-class classifiers of varying complexity and performance are experimented with, namely K-Nearest Neighbors (KNN) and Support Vector Machine (SVM). To perform outlier screening, the probability estimation available in the SVM is leveraged. Given a sample, the SVM provides not only the chosen class, but also a vector containing the probabilities that this sample belongs to each known class. The *conjecture* of the outlier detection method is that when the sample comes from a known distribution (i.e., previously seen), the probability of the winning class will dominate all others, while when it comes from an unknown distribution (i.e., outlier), multiple classes will exhibit fairly similar probability. Therefore, a simple outlier screening criterion is the probability difference between the first and second most likely classes. If this difference exceeds a threshold, which can be learned through cross-validation, the process is classified as an outlier. In this implementation, KNN from the Matlab library and SVM from the LIBSVM library are used [11].

4.1.3 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.

The experiments were performed in Simics, wherein we simulated a 32-bit x86 machine with a single Intel Pentium 4 core running at 2GHz and containing 4GB of RAM, on which we loaded a minimum installation Ubuntu server that embeds a Linux 2.6 kernel, as our operating system. All collected data is normalized and fed to the analysis software via Python/Matlab.

Table 4.1: Process classification accuracy (subset of classes)

application class	training samples	testing samples	KNN accuracy	SVM accuracy
overall	2386	2376	96.97%	96.63%
bash	1088	1087	100%	100%
cjpeg	25	25	100%	100%
djpeg	25	25	96%	100%
susan	75	75	100%	100%
search	50	50	98%	98%
madplay	50	50	96%	96%
tiff2bw	50	50	98%	94%
tiff2rgba	50	50	100%	100%
tiffmedian	50	50	96%	100%
basicmath	50	50	92%	90%
toast	50	50	96%	96%
untoast	50	50	94%	94%
rawcaudio	25	25	92%	92%
rawdaudio	25	25	52%	52%
		.....		
run-parts	18	18	83.33%	83.33%
date	15	15	86.67%	86.67%
dpkg	11	11	72.73%	72.73%
savelog	9	9	55.56%	55.56%
cron	4	3	66.67%	66.67%
cmp	3	3	33.33%	33.33%

Process Classification Accuracy

To evaluate the accuracy of our method in correctly classifying processes, we use MiBench [24], a free commercially representative benchmark suite as our workload, which contains a few tens of application classes. The entire suite was executed 100 times, with each application invoked with various valid arguments or in the background. We also randomized workload execution to avoid the bias that a specific order might impose. We exploit the Simics feature, *haps*, to hook our event monitor on the iTLB and the program counter. Our feature extraction method was then applied on the workload log. In total, we collected a dataset

containing 4762 samples, each comprising a feature vector matrix and representing a process to be classified. Initial dimensionality of the feature vector matrix was as large as 83612 and was reduced to 200 after applying PCA. The reduced matrix was then fed into the two classifiers. Half of the samples of each application class were used for training and the other half for testing. The process classification results using KNN and SVM are shown in Table 4.1. As may be observed, both classifiers performed very well in correctly classifying the processes, reaching an overall classification accuracy of 96.97% and 96.63% respectively. For most classes, this accuracy was even higher. However, parasite processes such as `savelog`, `cron`, and `cmp`, can be created sporadically during the execution of MiBench applications in our simulation environment. Samples of these processes were considered in our experiments but their low frequency of occurrence limits the available samples and undermines the corresponding classification accuracy. Fortunately, considering their weight, their overall impact on process classification accuracy is small.

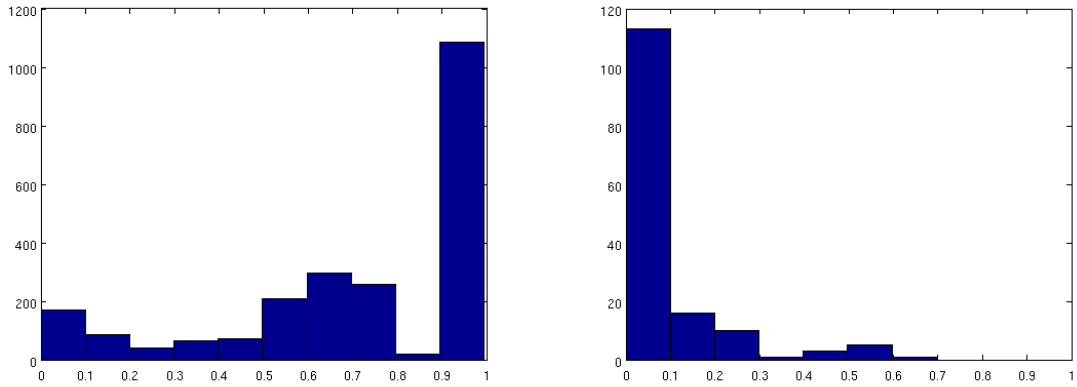
A noteworthy exception is the process `rawdaudio`, for which half of the instances are misclassified as `rawcaudio`, despite the adequate number of training/validation samples. This is explained by the fact that `rawcaudio` implements an Adaptive Differential Pulse Code Modulation (ADPCM) encoding algorithm,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

Outlier Detection Accuracy

To perform outlier screening, we leverage the probability estimation available in the SVM. Given a sample, the SVM provides not only the chosen class, but also a vector containing the probabilities that this sample belongs to each known class. The *conjecture* of our outlier detection method is that when the sample comes from a known distribution (i.e., previously

Table 4.2: Summary of FP and FN rates in outlier detection

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



(a) Seen processes (b) Outlier processes

Figure 4.4: Probability difference between top two classes

seen), the probability of the winning class will dominate all others, while when it comes from an unknown distribution (i.e., outlier), multiple classes will exhibit fairly similar probability. Therefore, a simple outlier screening criterion is the probability difference between the first and second most likely classes. If this difference exceeds a threshold, which is learned through cross-validation, the process is classified as an outlier.

To evaluate the effectiveness of our system in identifying previously unseen processes, we repeated the experiment, this time omitting 5 randomly selected classes from the training set, while retaining them in the testing set to mimic outlier processes. Through cross-validation, we set the threshold for outlier screening to 0.6 and we applied it to the processes in the testing set. Table 4.2 summarizes the results for four different runs. For each run, we

report the number of seen and outlier processes in the test set, as well as the false positive (FP) (i.e., seen process classified as outlier) and false negative (FN) (i.e., outlier classified as seen process) error rates. As may be observed, even the simple outlier screening method described above results in high outlier detection accuracy, with the average FP and FN values at 12.31% and 5.13%, respectively. This effectiveness is explained through Figure 4.4, which confirms our conjecture. Indeed, for previously seen processes, the probability difference between the top two classes is overwhelmingly high, while for outlier processes it is overwhelmingly low. Threshold adjustment can support biased decisions, favoring one error direction, while advanced outlier detection methods can further improve the results.

Logging Overhead

To evaluate the overhead of our method, we focus on its major hardware component, namely storage, and we seek to assess the required data logging rate. Unfortunately, Simics is not a cycle-accurate simulator. Therefore, to attain a more accurate estimation, we calculated the logging rate as follows. For each partition of a process, our method requires one feature vector containing 18 elements. If we assume `partition_size` to be 100, as in our experiments, we only need 7 bits for each element, since the occurrence frequency can never exceed the `partition_size`. The number of partitions per second for which a vector needs to be logged is determined by the iTLB miss rate. Assuming clock cycles per instruction (CPI) has an optimal value of 1, the estimated logging rate is calculated step by step by the equations below:

$$F.V.\ size = 18 \times \lceil \log_2 partition\_size \rceil \tag{4.1}$$

$$partition\ generation\ rate = \frac{iTLB\ miss\ rate}{partition\_size} \tag{4.2}$$

$$bits/inst. = F.V.\ size \times partition\ generation\ rate \tag{4.3}$$

$$est.\ logging\ rate(bits/sec) = \frac{bits/inst. \times clk\ freq.}{CPI(assumed = 1)} \tag{4.4}$$

We ran our benchmark suite several times to obtain an average iTLB miss rate, the value of which was 0.0016%, resulting in an estimated data logging rate of only 5.17 KB/sec. While a typical TLB miss rate is expected to be around 0.01-1% [38], since we consider only user-space virtual addresses and only iTLB misses, the relevant miss rate for our scheme is much less. Furthermore, since we assumed an optimal CPI of 1, the logging rate ought to be even lower in realistic cases. As a point of reference, the performance counter-based method in [19], which performs similar analysis with a different objective (i.e., malware detection vs. workload forensics), requires bandwidth of a few hundred KB/s.

4.1.4 Conclusion

In this work, a hardware-based approach is proposed for performing workload reconstruction and process identification for the purpose of forensic analysis. Unlike OS-level and hypervisor-level methods, which rely on information obtained through the OS and are, therefore, vulnerable to software attacks, this hardware-based method extracts and logs the required information directly in hardware, making it impervious to such attacks. Herein, a simple incarnation of this general idea was demonstrated, which relies on identifying instructions causing an iTLB miss and extracting/logging appropriate features, based on which a statistical analysis can, then, perform process identification. The proposed method was evaluated on a 32-bit x86 architecture running Linux OS, which was implemented in the Simics simulation environment, alongside a statistical analysis module which employed KNN and SVM for the purpose of process classification. Experimental results using the popular Mibench benchmark suite reveal that an overall process classification accuracy of 96.97% can be achieved with very low logging rate. Nevertheless, the performance of the identification of some program classes can be potentially improved, which may required advanced machine learning algorithm.

4.2 Off-line Workload Forensics using Temporal Features

4.2.1 System Overview

In this work, an advanced framework, i.e., TLB profiling Expert (TPE), is proposed, in order to address the limitation in the work above and improve the performance further. The TPE, similar to the work above, follows the off-line system architecture, consisting of two main components, i.e., *logging module* and *analysis module*, while the feasibility of workload forensics using temporal features (rather than spatial features) mined from the TLB profile (i.e., instructions raising iTLB miss) is explored. This idea is motivated by the fact that temporal features, compared with its spatial counterpart, can convey more information regarding the program behavior, and therefore, may potentially improve the performance in identifying workloads. Furthermore, 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 purpose. 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.

4.2.2 Implementation

Logging Module

The logging module in TPE collects data exclusively from the hardware related to the process identifier, and performs feature extraction in order to generate representative features for modeling the program behavior. In x86, as mentioned before, the `CR3` value can be used as a process identifier. Similarly, the RISC-V architecture cooperates with the feature of page virtualization as well and maintains a page table which facilitates the translating 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.

In order to extract the temporal features, we first pre-process the logged data to obtain an abstraction of its semantic. Upon the pre-processed data, three types of features, i.e., counts of occurrence with partitioning, n-gram model and raw sequence of categorized operator, are then developed and evaluated. The counts of occurrence feature is the same as proposed in the last chapter and is used as a reference herein, while only the latter two features will be discussed in this section. Moreover, the same methodology is shared between the x86 and RISC-V architecture, while slight difference is involved in data pre-processing due to the distinction between the x86 instruction set and RISC-V instruction set. The actual feature extraction procedure is introduced as follows.

The semantic of the logged instruction sequence is first abstracted 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.

On the other hand, 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.

Regarding the RISC-V architecture, 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 [44]. Alternatively, 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 five types of operators as follows:

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 the process behavior on RISC-V architecture can then be developed.

The two temporal features, i.e., n-gram model and the raw sequence model, can then be constructed as follows. 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 so that it cannot be directly used on our logged instruction sequences, which are multivariate sequences, containing variables of both operators and operands.

Current researches 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 argument [30, 46, 9]) while their effectiveness has been verified. Similarly in our scheme, the operators can be expected to convoy 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

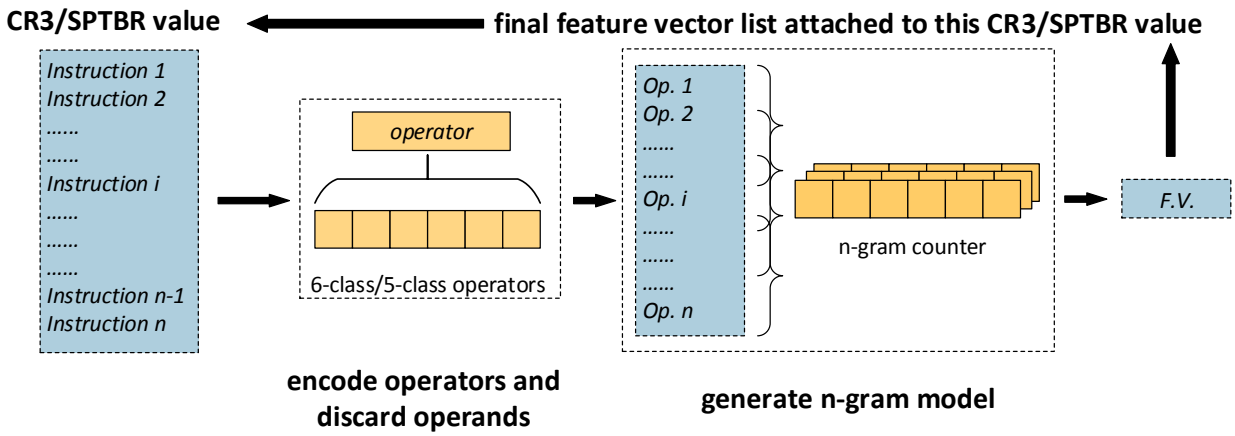


Figure 4.5: Feature extraction - n-gram

the original multivariate instruction sequence can then be converted into the univariate operator sequence, on which the n-gram model can be easily applied, as shown in Figure 4.5. As a result, feature extraction using n-gram model, compared with number of occurrence with partitioning, maintains frequency and lossy order information with a feature matrix of significantly reduced size whereas it ignores operands information.

A n-gram model tries 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 sieved 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 expects *independent features* in the feature vector cannot accept sequential inputs. Hence, it is necessary to employ more advanced machine learning algorithm, i.e., deep learning model, in order to process the *sequential features*.

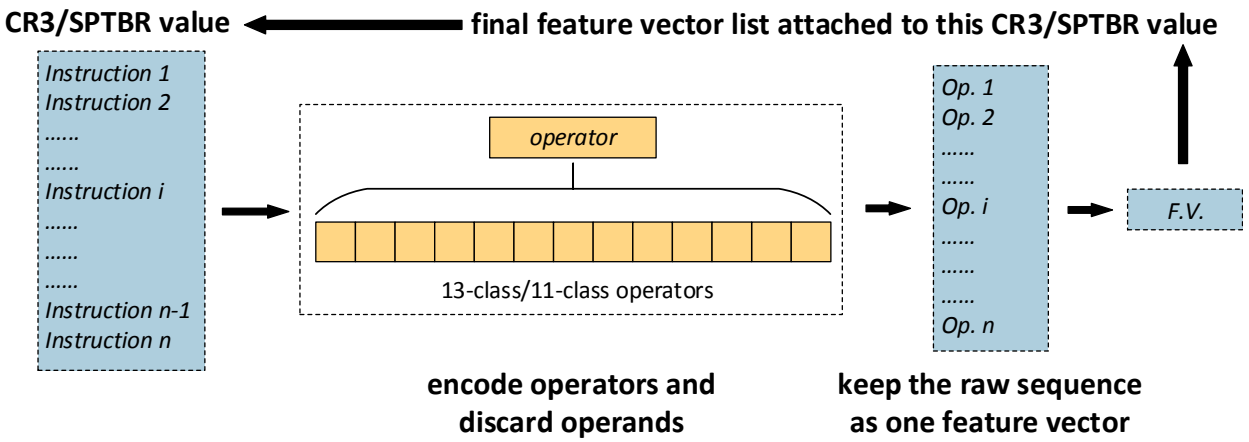


Figure 4.6: Feature extraction - raw operator sequence

Deep learning is a branch of machine learning, which attempts to model high-level abstraction of data through multiple processing layers. Using deep learning model benefits us in two ways. Firstly, certain architecture of the deep learning model can process sequential inputs so that it is a perfect complement to our sequential features. On the other hand, deep learning algorithm 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 sequence, as well as the assumption that the operator is more informative in program behavior modeling, the actual feature we use is the operator sequence while operand information is discard, as shown in Figure 4.6. Furthermore, after learning the log of the instruction sequence, it is revealed that most instructions fall into **DATA Op.** set and **ALU Op.** set. Hence, for both the x86 and RISC-V scenarios, we extend the categories of these two operators as illustrated in Table 4.3. Consequently, 13 classes of operators, rather than the original 6 classes, are evaluated alternatively for the x86 scenario, while 11 classes of operators, rather than the original 5 classes, are evaluated alternatively for the RISC-V scenario.

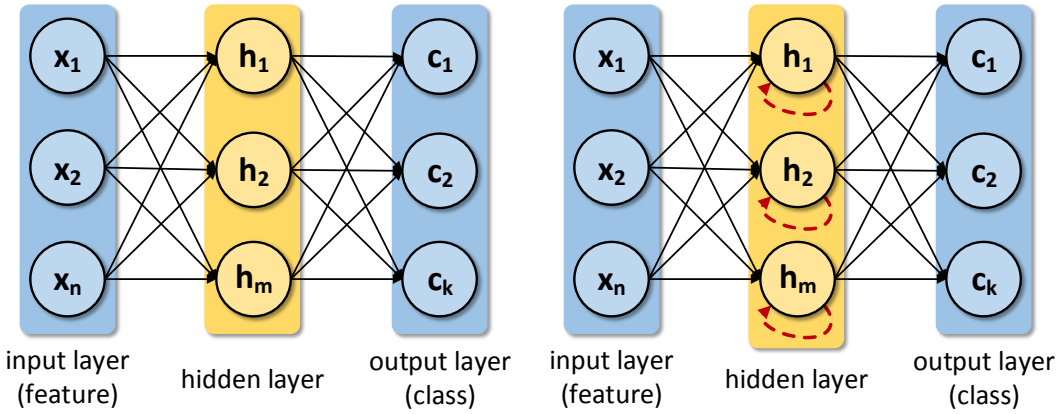
Table 4.3: Extended classes of DATA Op./ALU Op. set for both x86 and RISC-V scenario

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

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 early stage so that a subsequence of their execution flow can be sufficient to provide distinguishable features [16]. In this work, the *early prediction* effect was evaluated on the temporal features introduced above. Specifically, the corresponding feature extraction mechanism is performed only on a fixed-length subsequence of the operator sequence. Various possible lengths of the subsequence are experimented exhaustively while the optimal length is selected based on the statistical result. Apparently, the *early prediction* effect simplifies the feature extraction mechanism in hardware, leading to significant decrease in the memory overhead, as well as minimizing the size of feature matrices so that the computational complexity of the further forensics analysis can be reduced.

4.2.3 Analysis Module

Similar to the prior work, the analysis module is implemented in a trusted software environment, based on machine learning algorithms which facilitating multi-class classification. For the purpose of process reconstruction, Three different non-linear multi-class classifiers



(a) Architecture of ANN (b) Architecture of RNN

Figure 4.7: ANN vs. RNN

of varying complexity and performance are experimented with, namely K-Nearest Neighbors (KNN), Support Vector Machine (SVM) and Recurrent Neural Network (RNN). KNN and SVM, as traditional machine learning algorithm, are employed to handle the n-gram model in the same manner as the prior work. RNN, on the other hand, as the more advanced deep learning algorithm, is employed to handle the raw sequence model. In our implementation, we used KNN from the Matlab library and SVM from the LIBSVM library [11].

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 user-defined number of nodes, i.e., *neurons*, who 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 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 4.7a. Apparently, ANN evaluates each feature element independently, and therefore, cannot process sequential features.

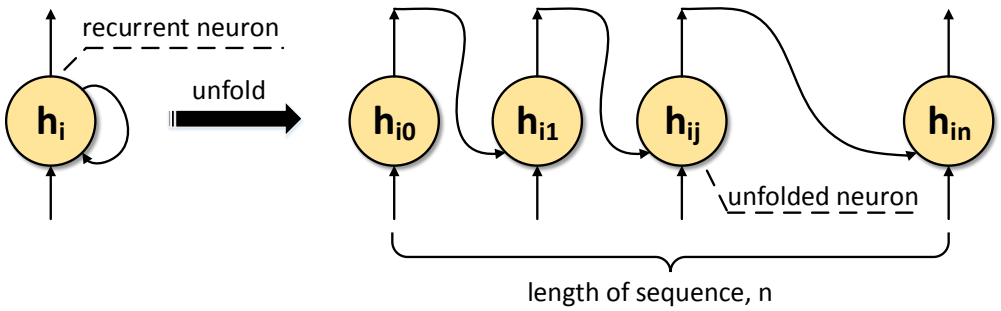


Figure 4.8: An unfolded recurrent neuron in RNN

RNN, on the other hand, considers the sequential information through simple modification on the traditional ANN. Specifically, in RNN, a self-feedback is applied on each neuron so that its outputs, now, not only rely on inputs from the last layer but also depend on the previous computations of its own. For better understanding, a RNN can be converted into the traditional ANN through unfolding the feedback of its neurons, as shown in Figure 4.8, while the depth of the unfolded network depends on the length of the input sequence. By this mean, RNN *memorizes* information of what has been calculated, and therefore, leverages the sequential information in the input sequence. A typical architecture of RNN is illustrated in Figure 4.7b.

The conventional method to train an ANN, 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 *vanishing gradient* problem, as identified in [6]. 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, thereby, 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 working at all. The RNN, unfortunately,

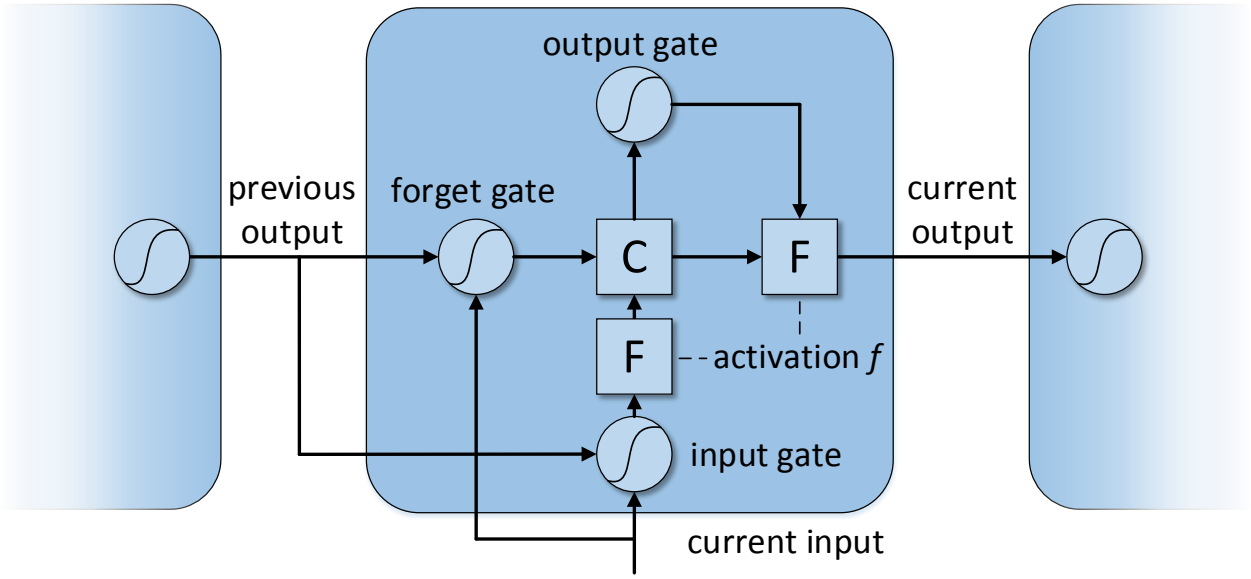


Figure 4.9: The implementation of the memory cell in LSTM-RNN

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 [25]. LSTM-RNN substitutes the original neuron with a *memory cell*, whose implementation is shown in Figure 4.9. A memory cell generally consists of an input gate, a neuron with self-feedback, a forget gate and an output gate [23]. 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 it. The forget gate, on the other hand, controls the effect of the previous memory state [25]. By this mean, LSTM maintains a more constant error propagation during BPTT training, which enables the RNN to learn over much longer steps, thereby prevents the vanishing gradient. In our implementation, we used LSTM-RNN from Keras [13].

In order to identify unseen processes, which should not be classified as any existing process class, the outlier detection is performed. Specifically, two different approaches are

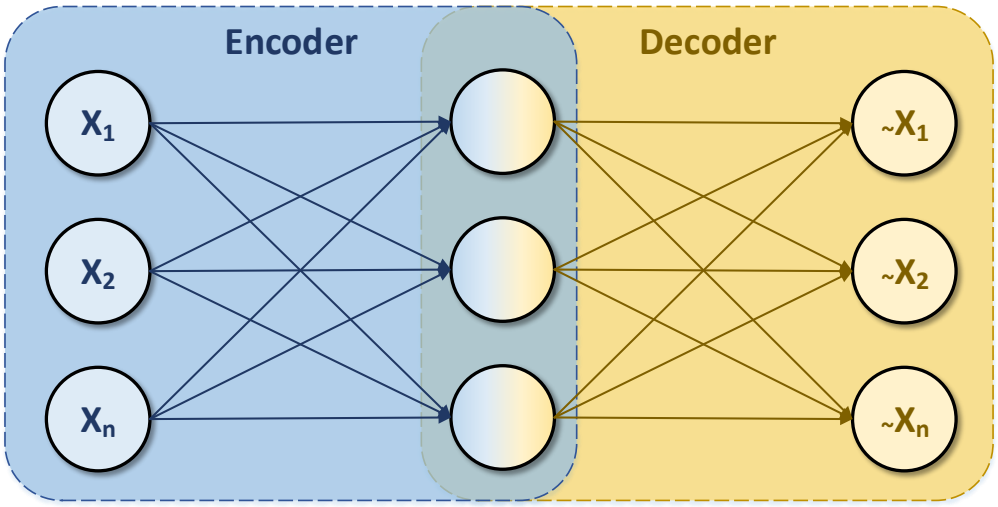


Figure 4.10: Typical architecture of an auto-encoder

considered, i.e., probability estimates and auto-encoder, to distinguish unseen process from seen process. The former is the same as introduced in the prior work, which is used a reference herein, while the latter is discussed in detail as follows.

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 4.10.

In this work, in order to enable the compatibility with sequential inputs of the auto-encoder, the LSTM-RNN is employed as the network architecture. Particularly, it is attempted 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 number of abnormal reconstruction error as follows:

$$e_i = \begin{cases} e_{abr}, & if\ e_i > e_{max}(i) \\ e_{norm}, & \text{otherwise} \end{cases} \tag{4.5}$$

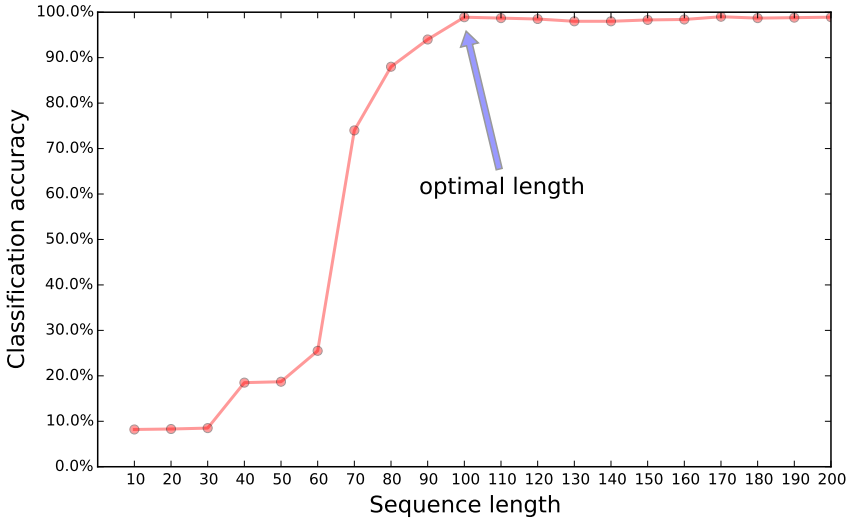
$$sample = \begin{cases} seen\ proc., & if\ |\{e_i \mid e_i = e_{abr}\}| < th \\ outlier, & \text{otherwise} \end{cases} \tag{4.6}$$

Considering the underlying distribution describing different seen process classes may vary significantly, in this work, we build auto-encoder for each seen process classes while their corresponding thresholds are set separately.

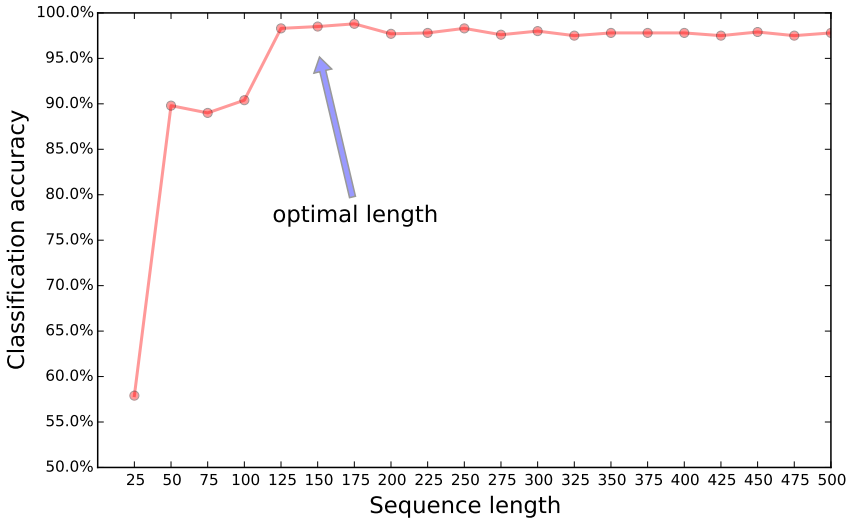
4.2.4 Experimental Results

The experimental setup remains the same as in the prior work for x86 architecture. On the other hand, The experiments for RISC-V architecture 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 miss and the change of SPTBR value.

Similarly, the MiBench benchmark was used as the workload, while, 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. Consequently, the rest of the benchmark suite were executed 400 times in the same manner as in our previous experiments, which results in approximately 4800 samples in total. Similarly, each samples represents a single process.



(a) Classification accuracy over different input sequence length on x86



(b) Classification accuracy over different input sequence length on RISC-V

Figure 4.11: Early prediction analysis on x86 and RISC-V

Early Prediction Analysis Using LSTM-RNN

The early prediction effect was first evaluated in the process reconstruction using LSTM-RNN approach on both x86 and RISC-V architectures. We note that LSTM-RNN accepts raw operator sequence as its inputs 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 evaluated length of the operator sequence varied from 10 to 200 on x86 while it varied from 25 to 500 on RISC-V. Figure 4.11 summarized 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 lengthened, which corresponds to the fact that longer sequence convoys more information for better distinguishability. On the other hand, the input length longer than 100 on x86 has no significant impact on the classification accuracy, which justifies the early prediction effect, indicating that process reconstruction based on the entire program execution flow is unnecessary at all. Similarly, such 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 as the optimal length of the input sequence on x86 as well as 150 as the optimal input sequence length on RISC-V.

Process Classification Result on x86

The process classification was performed on samples of 23 classes of processes, where each class had 400 samples. The dataset was split by half into training set and validation set, each of which contains half of the samples for every process class. Upon the two dataset, we compared the effectiveness of using the three types of features combined with three different machine learning model to identify different processes, as introduced in Section 4.2.2, which is illustrated in Figure 4.12.

We first evaluated the classification performance using the counts of occurrence features as a reference point. The initial dimensionality of the collected feature vector matrix was as large as 83612 and was reduced to 200 after applying PCA. The reduced matrix was then fed into the two classifiers, i.e., KNN and SVM, leading to the process classification results as

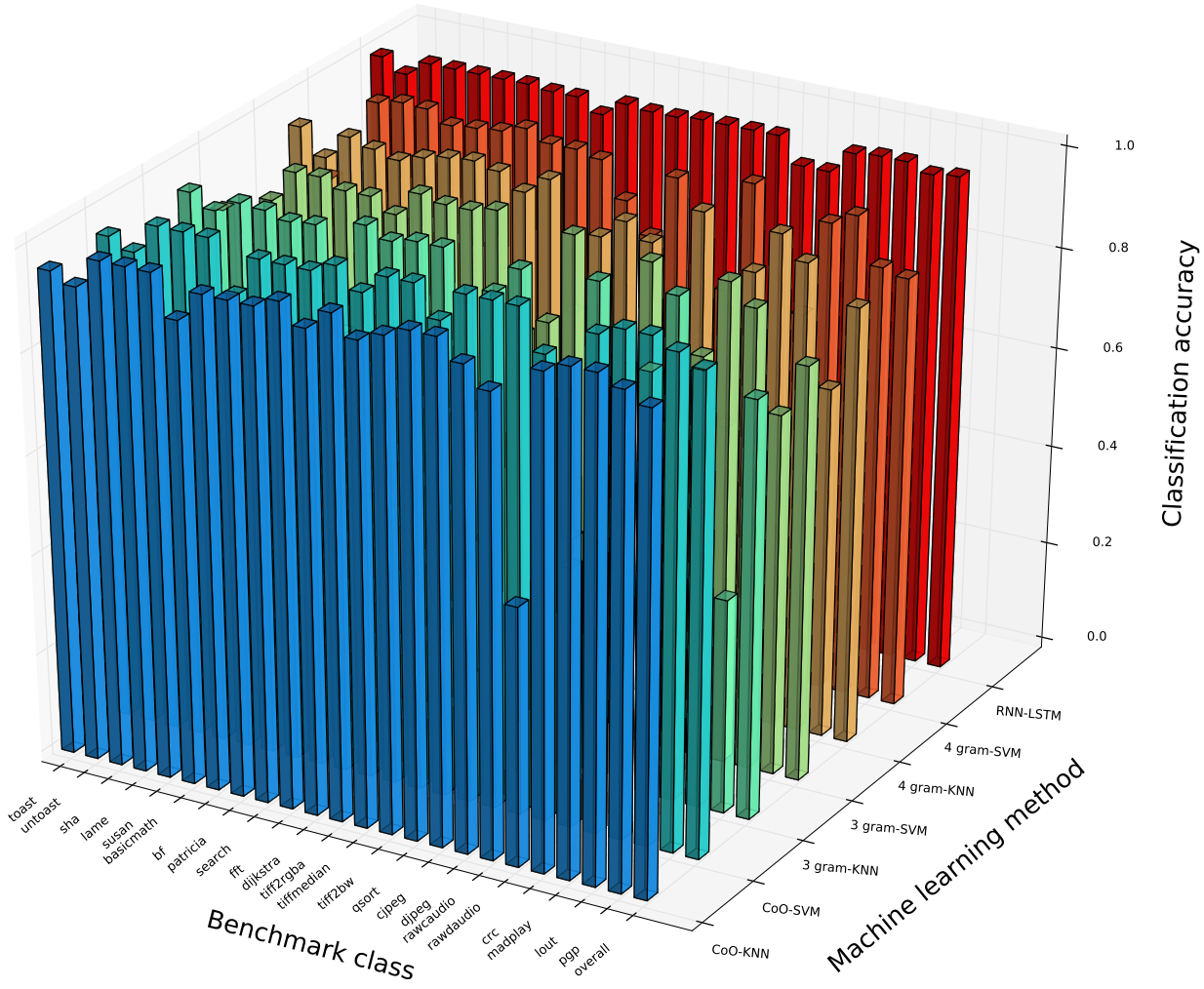


Figure 4.12: Process identification results on x86

illustrated by the first two rows (the blue row as the first) in Figure 4.12. The classification result accords to the observation in the prior work. Both classifiers performed very well in correctly classifying the processes, reaching an overall classification accuracy of 95.74% and 95.83% respectively while the process `rawdaudio` is an exception. Alternative features with more advanced machine learning algorithms, as proposed in this work, may potentially address this limitation.

The alternative features using n-gram model were evaluated next. We experimented with the 3-gram and 4-gram models, which were applied on the operator sequence of length 100 to extract features since the early prediction effect was revealed. As described in Section 4.2.2, since 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 accordingly. Compared with the case of using counts of occurrence, the dimension of the feature matrix generated using n-gram model was far less. Similarly, the matrix was then fed into KNN and SVM to perform the process classification. As indicated by the third to the sixth rows in Figure 4.12, 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%. As expected, The 4-gram model, since it captures information in finer-grained manner, performed better than the 3-gram model. However, compared with the result of using counts of occurrence, the overall performance of the n-gram model was not competitive. Neither the issue between the process *rawcaudio* and *rawdaudio* was 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 while they sacrificed potentially helpful information from the operands. Nevertheless, the n-gram model generated a much smaller feature matrix, which implies dramatically reduced storage/computation overhead.

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 equals to the optimal length of the operator sequence, i.e., 100, due to the early prediction effect. As shown by the last row (the red row) in Figure 4.12, the RNN-LSTM model performs the best in process identification, achieving an average classification accuracy of 99.12%, 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 while 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.

Outlier Detection Result on x86

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

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 4.13 summarizes the results for four different runs. 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) rates. As may be observed, even the simple outlier screening method described above results in high outlier detection accuracy, with the average FP and FN rate at 12.31% and 5.13%, respectively. This result accords to the observation in the prior work and is used a reference point.

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, 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 4.14. The left y axis

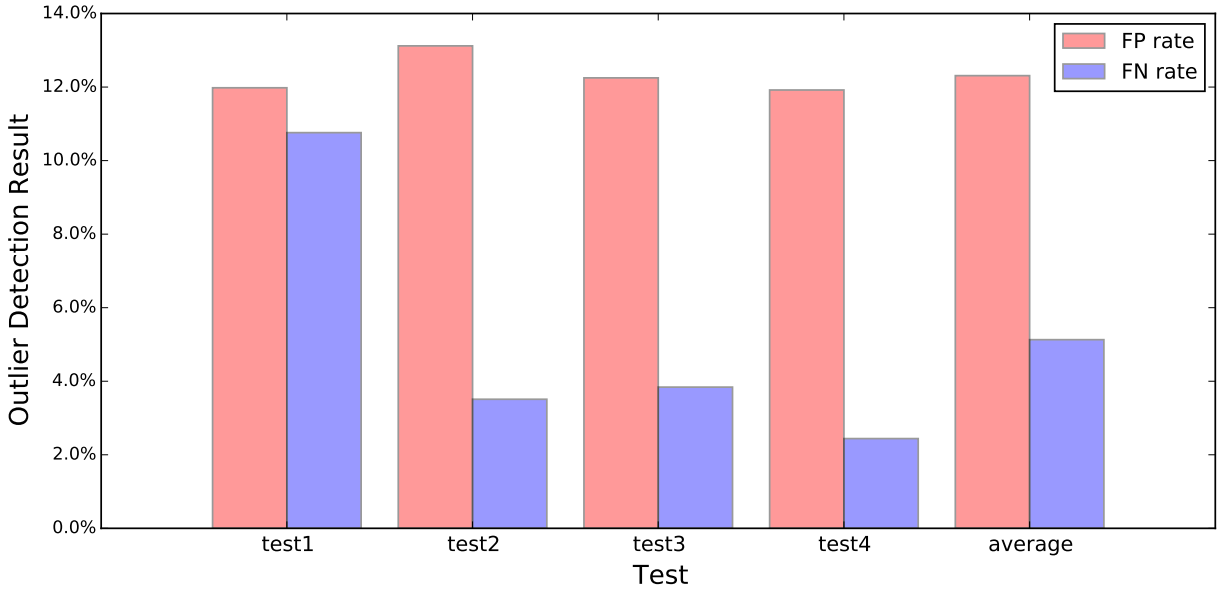


Figure 4.13: Outlier detection results using probability estimates on x86

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, which 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. 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 result. As a result, outlier detection using auto-encoder can be expected to outperform another alternative using probability estimates.

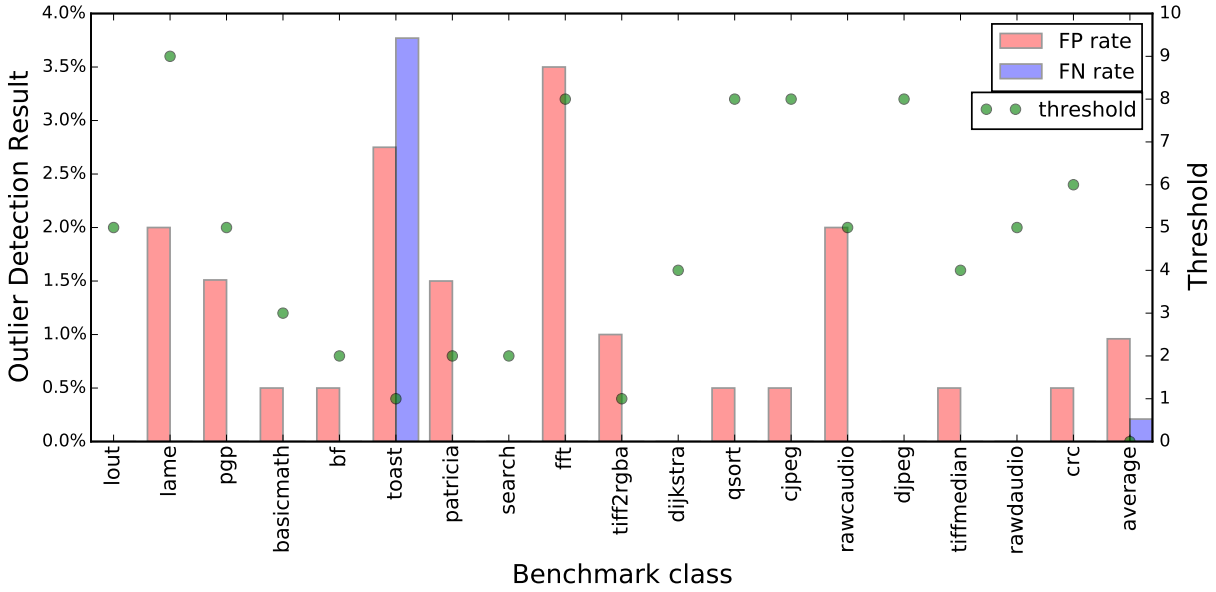


Figure 4.14: Outlier detection results using auto-encoder on x86

Logging overhead on x86

To evaluate the design overhead of the TPE on x86, we focus on assessing the required data logging rate corresponding to our different feature extraction mechanisms. 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:

$$F.V.\ size = 18 \times \lceil \log_2 partition\ size \rceil \tag{4.7}$$

$$partition\ generation\ rate = \frac{iTLB\ miss\ rate}{partition\ size} \tag{4.8}$$

$$bits/inst. = F.V.\ size \times partition\ generation\ rate \tag{4.9}$$

$$est.\ logging\ rate(bits/sec) = \frac{bits/inst. \times clk\ freq.}{CPI(assumed = 1)} \tag{4.10}$$

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 the number of categories of operators to be 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:

$$element\_size = \lceil \log_2 number\ of\ op.\ categories \rceil \tag{4.11}$$

$$bits/inst. = iTLB\ miss\ rate \times element\ size \tag{4.12}$$

$$est.\ logging\ rate(bits/sec) = \frac{bits/inst. \times clk\ freq.}{CPI(assumed = 1)} \tag{4.13}$$

We ran our benchmark suite several times to obtain an average iTLB miss rate, the value of which was 0.0016%, resulting in an estimated data logging rate of only 5.17 KB/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% [38], 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 scheme are only enabled for the first 100 instructions

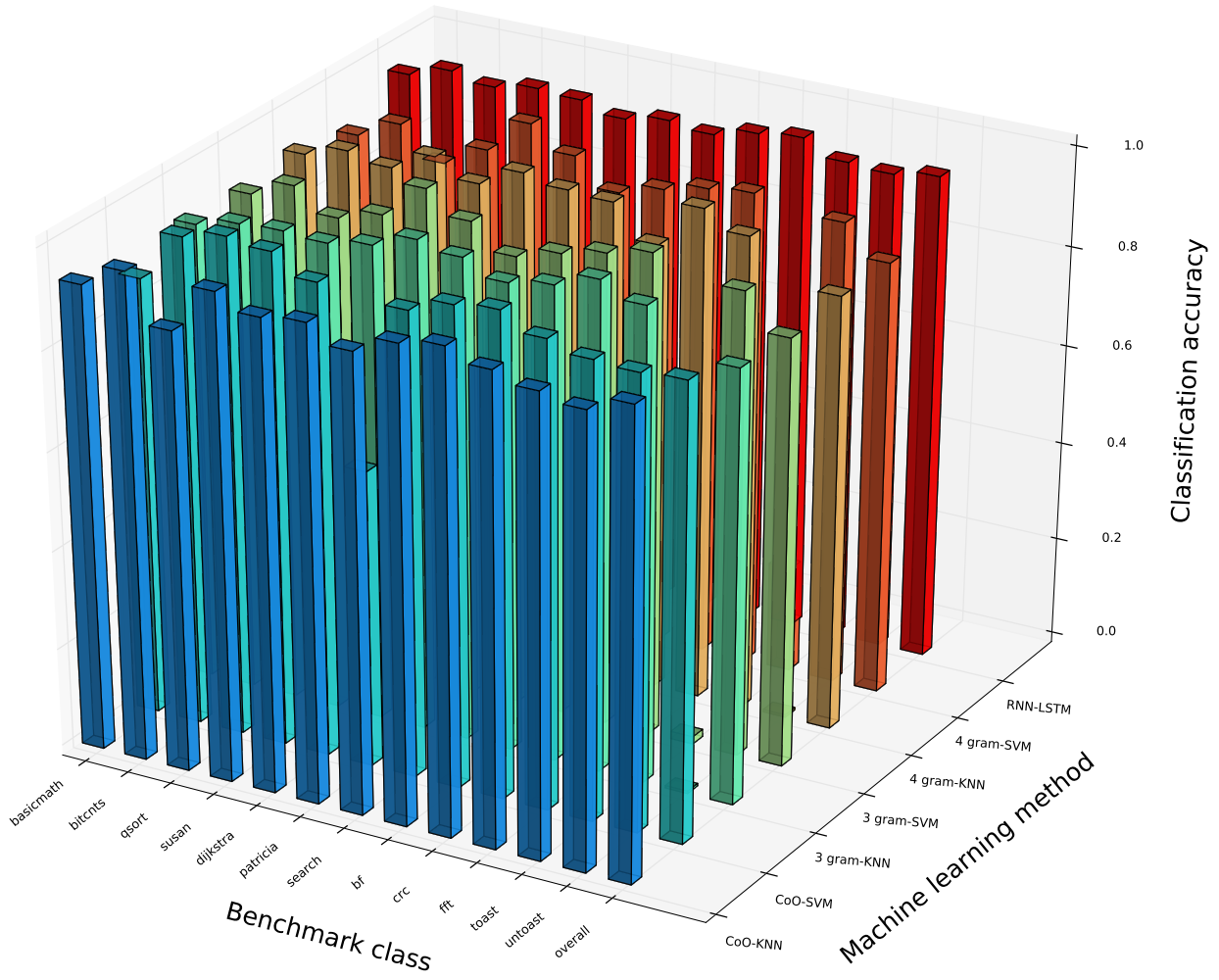


Figure 4.15: Process identification results on RISC-V

raising iTLB miss while remains disabled during the rest of time, hence, generating logs of much smaller size. Therefore, n-gram model and raw operator sequence, although introduce higher rate during the logging process, they, in fact, incur less storage overhead.

Process Classification Result on RISC-V

This section evaluates the effectiveness of the TPE in process identification on RISC-V. The experiments were performed on samples of 12 process classes, where each class contained 400

samples. The dataset was split by half into training set and validation set, each of which contained half of the samples for every process class. The same experiments as in the x86 scenario were performed on the two datasets, whose results are illustrated in Figure 4.15.

We start with evaluating the classification performance using counts of occurrence. Similarly, a first-level dimensionality reduction using PCA was performed on the collected dataset. The compressed dataset was then fed into the KNN and SVM model, engendering the classification results as illustrated by the first two rows in Figure 4.15. As may be observed, both classifiers achieved similar results in process identification as in the x86 scenario, reaching an overall accuracy of 94.5% and 92.1% respectively. Alternative features and machine learning algorithms may further improve the results.

The 3-gram and 4-gram models, which were applied on the operator sequence of length 150 for feature extraction, were evaluated next. Upon the 5 types of operators as mentioned in Section 4.2.2, the 3-gram and 4-gram model generated the feature matrices whose initial dimensionality were 125 and 625 respectively. The feature matrices were then fed into KNN and SVM to perform process classification. Accordingly, as illustrated by the third to the sixth rows in Figure 4.15, 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. Likewise, the n-gram model does not bring advantageous results in process identification over the counts of occurrence, while it is beneficial in reducing the massive storage/computation overhead introduced by the latter.

Finally, we evaluated the performance of the deep learning model using the raw operator sequence in process identification on RISC-V. The dimensionality of the feature matrix in this scheme is 150, which accords to the optimal length of input sequence, achieved by the early prediction analysis. As shown by the last row in Figure 4.15, the RNN-LSTM model performs the best as well in process identification on RISC-V, reaching an overall classification accuracy of 97.8%, which is approximately 3% higher than the accuracy achieved

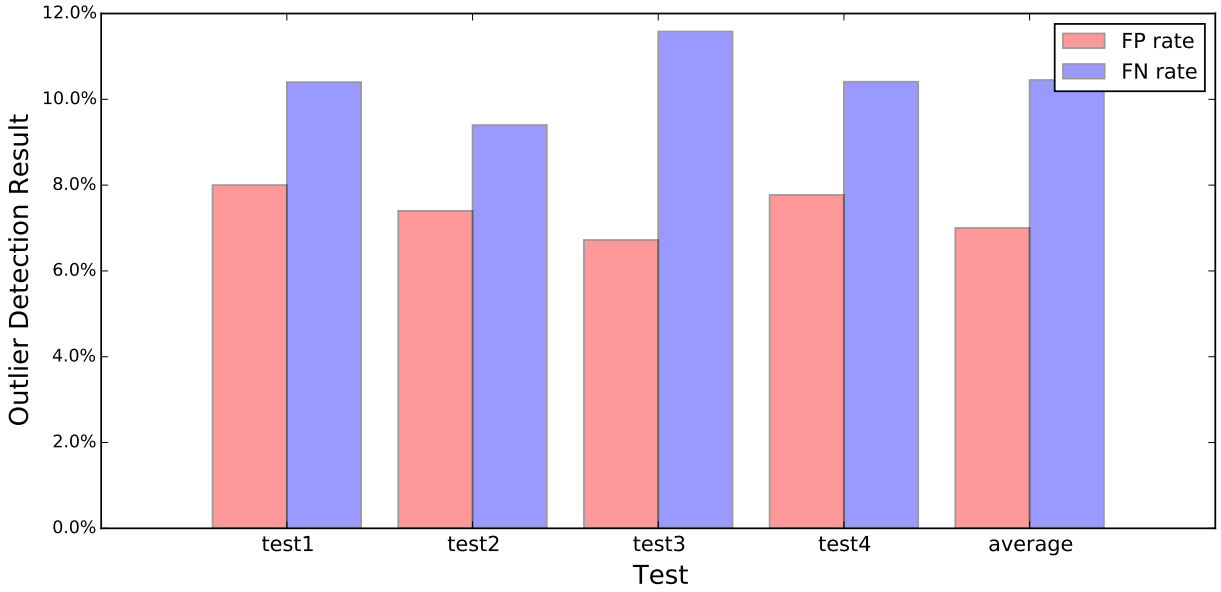


Figure 4.16: Outlier detection results using probability estimates on RISC-V

through KNN/SVM with counts of occurrence. In a nutshell, the TPE is able to perform well in identifying processes on both x86 and RISC-V architectures, while the RNN-LSTM model consistently achieves the best performance on both architectures. This observation, therefore, corroborates the effectiveness of the sequential features as well as the generalizability of the TPE.

Outlier Detection Result on RISC-V

Regarding evaluating the outlier detection on RISC-V, we follow the same experimental flow, as in the x86 scenario, i.e., omitting 5 randomly selected classes from the training set while retaining them in the validation set to mimic outlier processes. The performance of the proposed methods using probability estimates and auto-encoder was then evaluated as follows.

The outlier detection using probability estimates is first evaluated. The threshold for outlier screening in this case is set to 0.6 through cross-validation. As shown in Figure 4.16,

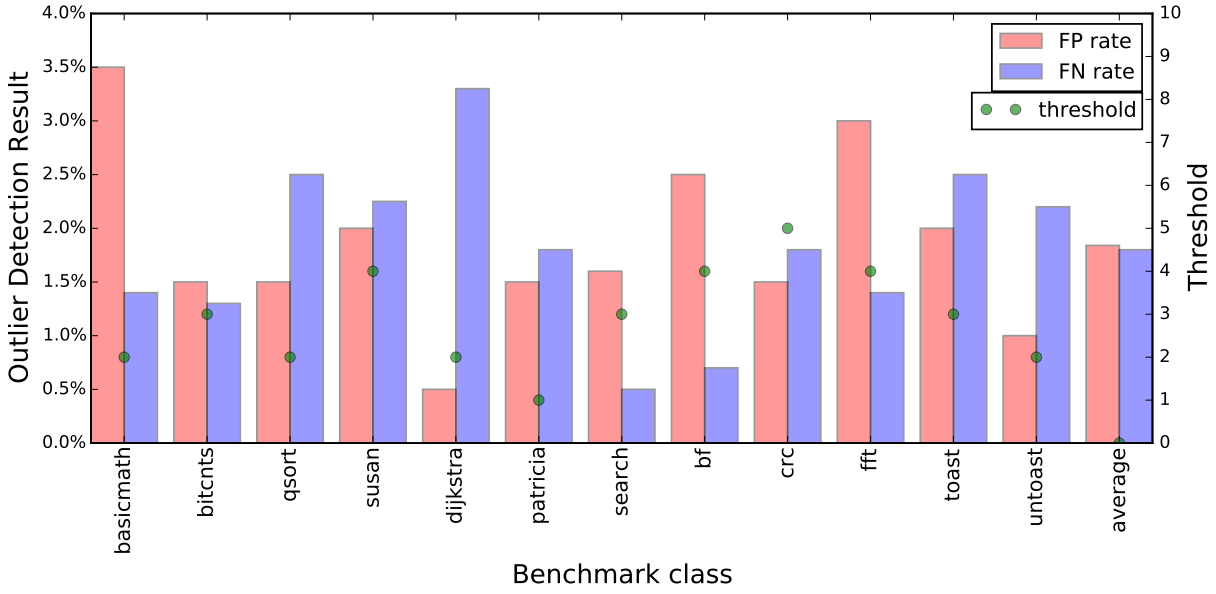


Figure 4.17: Outlier detection results using auto-encoder on RISC-V

the repeated tests indicate a high outlier detection accuracy, with the average FP and FN rate of 7% and 10.45% respectively. The consistent results in both x86 and RISC-V scenarios confirm the effectiveness of the proposed simple outlier screening method.

To evaluate the performance of the auto-encoder on RISC-V, the same experiments were performed, as in the x86 scenario. As a result, 12 independent auto-encoders were constructed with individual outlier screening thresholds. Figure 4.17 summarized the outlier detection results. As may be observed, an average FP rate of 1.84% as well as an average FN rate of 1.8% can be achieved, which is significantly lower than the counterpart in the case of using probability estimates. The worst case of the FN and FP rate is 3.3% and 3.5% respectively. This results corroborates the effectiveness of the auto-encoder on both x86 and RISC-V architectures.

Logging Overhead on RISC-V

Since Spike is not a cycle-accurate simulator, the data logging rate, similarly, has to be computed manually. On the other hand, although the data pre-processing on the RISC-V differs from its x86 counterpart due to the distinct instruction set design, the methodologies for feature extraction are identical on both architectures. As a result, the same formulas, i.e., equation (4.7) - (4.13), can be used in order to estimate the data logging rate of the TPE on the RISC-V.

In our experiments, 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 according to a RISC-V CPU prototype, i.e., SiFive E51, running at 1.5 GHz [1], an average data logging rate of 71.7 KB/s, 146 KB/s and 195 KB/s can be achieved when counts of occurrence, n-gram model and raw operator sequence are applied in feature extraction, respectively. The relatively high iTLB miss rate, compared with the case on 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 in parallel to this work, is expected to lower the iTLB miss rate.

4.2.5 Conclusion

In this work, the TPE was proposed as an extended work of the prior work. More specifically, the feasibility of using temporal features in hardware-based workload forensic analysis processed with advanced machine learning algorithms, i.e., RNN-LSTM and autoencoder, was explored. 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. Comparison between the performance of spatial features and temporal features 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 using temporal features.

4.3 On-line Workload Forensics

4.3.1 System Overview

In this work, an on-line workload forensics method is proposed. More specifically, we explore the possibility of relying exclusively on custom hardware components in order to trace architectural data of interest and further identify the executed workloads *in real time* using machine learning algorithms. The actual implementation consists of a hardware tracing module, a feature extraction module, and a workload identification module. The hardware tracing module is able to collect architectural events related to program execution exclusively from the hardware, whose data collection bus must remain invisible to OS-level applications. The feature extraction module generates representative features from the collected data, which describe program behavior, while the workload identification module wraps a machine learning-based classifier to identify (1) whether one workload is legitimate or not, (2) what a workload is if it is legitimate, through their dynamic behavior at the granularity of *process*. Herein, the machine learning algorithm is involved to consider the runtime variation of program execution. Due to the real time characteristic and the on-line system architecture, the

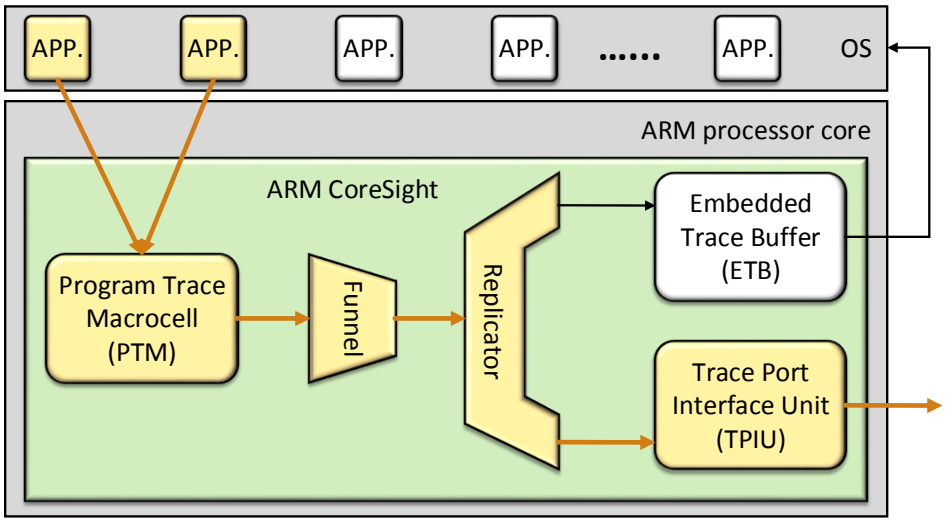


Figure 4.18: Architecture of ARM CoreSight

set of features and the machine learning algorithms to be used need to be carefully selected, since the complexity in the features and the algorithm may incur significant design overhead, which prevents the proposed method from deployment.

4.3.2 Implementation

HW tracing

The first and the foremost building block in our proposed framework is a hardware tracing component, which logs architectural events that can distinguish program behavior. Intuitively, the most informative event capable of modeling the program behavior is the dynamic control flow. Control flow tracing in hardware, however, is not straightforward at all, since it requires deep coupling with the execution pipeline of the underlying microprocessor while it is expected to introduce minimal performance overhead. Fortunately, industrial-standard hardware tracing solutions have been proposed, e.g., ARM CoreSight and Intel Processor Tracing (PT) [4, 29]. Generally speaking, these solutions aim at *non-intrusively* collecting program runtime *branch addresses* so that the dynamic control flow of a program can be

Table 4.4: Summary of application scenarios of commercial processors with hardware tracing support

Applications	**ARM**	**Intel**
Server and Desktop	Cortex-A75/A55	Xeon D family, Xeon E3/E5 family, Core i5/i7/i9 family
Mobile devices	Cortex-A73/A57, Snapdragon (Qualcomm), Ax (Apple)	Core i3/i5/i7 family, Core m3/m5/m7 family
Embedded applications	Cortex-A35/A17, Cortex-M23/M7/M4	Core i5/i7 family, Xeon E3 family

reconstructed with the assistance of the binary image of programs. For example, the ARM CoreSight employs a hardware macro, i.e., the Program Trace Macrocell (PTM) to fulfill the task. During the execution of an application in the OS, the PTM generates multiple types of packets according to customized trigger rules, which logs current context ID value, direct and indirect branch address, timestamps, etc. These packets are compressed in a specific way defined by ARM in order to minimize the bandwidth of the data log. The generated packets are then sent to the data storage through a communication channel, namely *funnel*. Two different data storage are introduced in CoreSight, namely Embedded Trace Buffer (ETB) and Trace Port Interface Unit (TPIU). The ETB maintains data in on-chip RAM so that software debug tools can later access it, while the TPIU drives the external pins of the trace port so that the trace data can be offloaded to an external hardware. Hence, we follow the latter path to collect our data of interest. An architectural view of the ARM CoreSight design is shown in Figure 4.18.

In order to simplify the design complexity as well as ensure the practicality of our workload forensics framework, we decide to take advantage of the state-of-the-art industrial-standard hardware tracing techniques. Considering the easier accessibility to the physical devices embedding ARM processor core and its hardware tracing module, the ARM Core-

Sight is employed in our proposed framework to facilitate the tracing task. Nevertheless, we note that the proposed framework is not ARM CoreSight-dependent. Essentially, any state-of-the-art hardware tracing solution, e.g., Intel PT, or custom solutions, can be plugged into this framework, while the ARM CoreSight is selected in this work only to facilitate the illustration of the proposed concept.

On the other hand, the wide adoption of the hardware tracing technology in the latest commercial processors eases the data acquisition of the proposed framework in various real-life application scenarios, as summarized in Table 4.4. For instance, both the ARM Cortex-A processor series, which bolster high-performance consumer infrastructure devices, and the Cortex-M processor series, which are optimized for low-cost Microcontroller (MCU) or System-on-Chip (SoC), are equipped with the ARM CoreSight solution. So does the Intel processor architecture, which embeds Intel PT starting from its 5th generation, i.e., the Broadwell in 2015.

In order to collect representative architectural data to describe program behavior and bridge the semantic gap, the ARM CoreSight is configured to trace the value of context ID register, which is interpreted as a *process identifier* [26], and the corresponding direct and indirect branch target addresses, which describe the program control flow and, thus, model the *program behavior*. Upon the trace collected through the CoreSight module, descriptive features are then generated in the feature extraction module.

Feature Extraction

Modeling program behavior using branch addresses exclusively is restricted by the intrinsic implementation of the ARM CoreSight. Nonetheless, the transition between branch addresses is considered to be sufficient to reveal both the static information of the execution of an arbitrary application, i.e., the layout of its address space, as well as the corresponding dynamic information, i.e., the program execution control flow. To facilitate the next-step

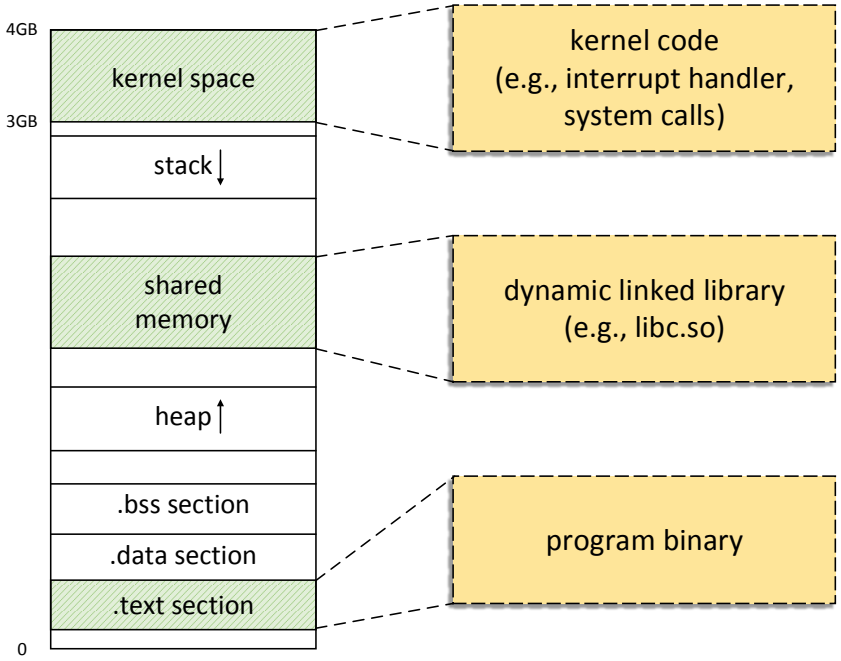


Figure 4.19: Typical address space layout

workload identification, the feature extraction component extracts descriptive features from the collected sequence of branch addresses. Specifically, we evaluate both the potential spatial features and the temporal features as discussed below.

We perceive the spatial features as the features which are able to capture the information of the address space layout of different applications. A common choice is, the Counts of Occurrence (CoO), which partitions the address spaces and then collects the counts of hits by branch addresses on each partition. Generally speaking, a finer-grained partitioning provides a more precise view of how an application organizes and utilizes its address space during runtime, while the size of the feature space grows linearly according to the granularity and incurs higher implementation overhead.

A typical address space layout is shown in Figure 4.19. As may be observed, the locations containing program runtime code mainly fall into three sections (i.e., **.text** section, **shared memory** section and the **kernel space** section), while the .text section maintains the user-

level code of the program, the shared memory section contains dynamically linked C shared library and the kernel space section keeps the OS service routines. This results in an extreme bias in the hit area in the address space of the branch target addresses. Therefore, splitting the address space evenly leads to a sparse feature vector and creates a lot of dummy entries, which remain zero or insignificant number and carry no useful information according to the increase of granularity. In contrast, we apply a weighted partitioning method, whose essential idea is to assign more partitions to the dense area (containing more CoO), while assigning fewer partitions to the sparse area (containing less CoO).

The partitioning problem can then be modeled as follows: given an address space `AS` and a target partition number `P`, find a set of edges `E` whose size is `P-1` and which partition the `AS`, so that the standard deviation of the `P-size` dataset after partitioning, where each partition `p` contains accumulated CoO, is minimized. Essentially, this is an optimization problem which can be solved through gradient descent algorithm. However, since the `AS` can only be split in order, we develop a computation-friendly heuristic algorithm to fulfill the partitioning task. First, we assume the size of the minimal dividable partition `U` is 2^{12} Bytes, which match the `4K` page size, in order to reduce the computational complexity. Initially, the `AS` is, therefore, evenly split into a list `L` consisting of $2^{32-12} = 2^{20}$ partitions. Given a target `P`, we run the following process in iteration until the partitioning is done: (1) compute the average CoO over the `L` according to `P`, (2) accumulate the CoO in each u_i until the sum reaches the average, (3) the accumulated u_i forms one p_j, (4) exclude p_j from `L` and go back to step (1). Listing 4.1 shows the pseudocode of the partition algorithm. Compared with gradient descent, the time complexity of this algorithm is $O(n)$, which is far more efficient.

Essentially, the spatial features introduced above extracts the spatial relationship of different branch target addresses and generates a lossy representation, i.e., the CoO after partitioning. However, it fails to capture the temporal relationship, which is the order of different branch target addresses and may also be helpful for identifying program behavior.

```
input: L[$2^{20}$] and P
output: E[P-1]

total = sum(L), p_accum = 0, i = 0, P_left = P;
//iterate over each minimal dividable partition
for u in L:
  mean = total/P_left;
  p_accum $\leftarrow$ p_accum + L[u];
  if p_accum > mean:
    //p contains only one u
    if p_accum == L[u]:
      E[i] $\leftarrow$ u;
      total $\leftarrow$ total - p_accum;
      p_accum $\leftarrow$ 0;
    else:
      //p contains multiple u,
      ensure p has the value closest to the mean
      if |p_accum-mean| >= |p_accum-L[u]-mean|:
        E[i] $\leftarrow$ u - 1; //exclude current u
        total $\leftarrow$ total - (p_accum+L[u]);
        p_accum $\leftarrow$ L[u];
      else:
        E[i] $\leftarrow$ u;
        total $\leftarrow$ total - p_accum;
        p_accum $\leftarrow$ 0;
    i $\leftarrow$ i + 1;
    P_left $\leftarrow$ P_left - 1;
    if P_left == 1: //the left u will form the last p
      break iteration;
return E;
```

Listing 4.1: Heuristic weighted partition algorithm

A popular feature extraction alternative to maintaining the temporal information of a dataset is the *n-gram* model. An n-gram is a subsequence of `n` items derived from a given sequence. A feature vector can then be constructed with the number of all the possible n-gram subsequences. When `n` is greater than 2, n-gram model can, thereby, preserve the sequential information, while such information is less lossy with larger `n`. The total number of features generated by an n-gram model can be bound by the number of possible elements in a given sequence `m` and the choice of `n`, i.e., m^n. The n-gram model in our scenario is then generated as follows: (1) split the address space into arbitrary `P` partitions using the algorithm in Listing 4.1, (2) given a `n`, the n-gram model calculates the CoO of the transition combination between any `n` partitions, (3) the size of the feature vector is P^n. Due to the underlying implementation cost, herein, we only consider the 2-gram model.

While the n-gram model compresses the temporal information in a lossy manner, the original sequence of branch addresses itself can be used as a feature vector in a lossless way. Herein, we explore the feasibility of using a *partition sequence* which is transformed from the original branch address sequence where each element is substituted with the partition it belongs to. Nevertheless, traditional machine learning methods, which expects independent features in the feature vector, e.g., our spatial features, cannot accept sequential inputs. Therefore, it is necessary to employ a more advanced machine learning algorithm, which is able to process sequential features.

Real-time Workload Identification

State-of-the-art program behavior modeling methods generally require a complete execution flow to perform further analysis, which prevents a real-time response. However, as shown in [17], program behaviors tend to deviate at an early stage of their execution. Therefore, in our scheme, it may be feasible to perform the real-time identification analysis using only a subsequence of the branch target address sequence, which implies more prompt response for identifying workload, rather than the *ex post facto* identification analysis.

Herein, we explore the possibility of using a header portion of the complete program execution profile in order to perform real-time workload identification analysis. Nevertheless, the header portion contains lossy information, which may undermine the effectiveness of the classifier in identification. We evaluate various lengths of the branch target addresses sequence to be used, in order to find the minimal length of subsequence required, leading to similar identification accuracy as the mechanism using a complete program execution profile. Given the truncated subsequence, spatial or temporal features introduced above can then be extracted.

Upon the aforementioned extracted features, our workload identification mechanism employs several machine learning algorithm to timely understand the workload being executed at the granularity of a process. In particular, the actual analysis is performed in two stages as follows. The first-level analysis employs the machine learning algorithm for anomaly detection in order to identify the unexpected suspicious workload beyond the legitimate workload set. The second-level analysis leverages multi-class classification to identify if legitimate workloads are executed according to the design specification and regulatory compliance.

Regarding the spatial features, considering the program behavior is generally not linearly distinguishable, we experimented with two non-linear classifiers of varying complexity and performance, namely Decision Tree (DT) and Artificial Neural Network (ANN). In our scheme, we evaluate DT from the Matlab library and ANN from Keras [13].

With reference to the temporal features, although the traditional machine learning algorithm can process the n-gram model, advanced learning algorithm must be involved in order to process the partition sequence. Herein, similar to the prior work, we employ Recurrent Neural Network (RNN), which has been developed to accept sequential inputs. In our implementation, we used LSTM-RNN from Keras [13].

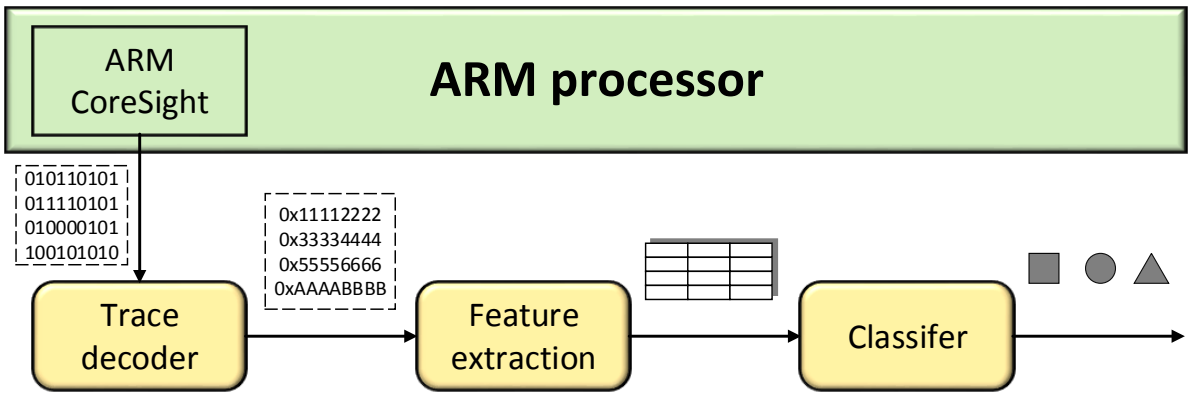


Figure 4.20: architectural view of the proposed framework

Hardware Implementation

In this Section, we present the hardware implementation according to the proposed framework. Since we leverage the ARM CoreSight to facilitate our design, it is unnecessary to design our custom hardware tracing module. Alternatively, a custom data decoder is required in order to decode the trace collected by the ARM CoreSight module. An architectural view of the entire framework is illustrated in Figure 4.20.

- **Trace decoder:**The trace decoder decodes the incoming data trace based on the packet format and the decoding rules introduced in the ARM CoreSight manual [4]. The decoder works at the same frequency as the ARM CoreSight module to synchronize itself with the CoreSight packet output. Only packets related to the current context ID value and the direct/indirect branch address, recognized by the predefined specific headers [4], are processed by the decoder, while the others are ignored. Upon the decoded branch address sequence, the next-level feature extraction can then be performed.

- **Feature Extraction:**This component extracts the features from the received branch address sequence in parallel with the data stream decoder, once it detects a valid decoded branch address. A feature vector is instantiated in order to store the CoO.

A conditional check is performed in order to access the correct partition based on the partition edge derived by the algorithm introduced in Listing 4.1. After the number of processed branch addresses reaches a pre-defined limit (which is determined through the evaluation in the following Section), this components finalizes the current round of the feature extraction.

Before the workload identification analysis is actually performed, the feature vector is standardized using the formula: $(X - \overline{X})/\Delta$, where X is the feature vector, and the coefficients $\overline{X}$ and Δ are the mean and standard variation vector derived from the training set. The standardization coefficients are pre-computed and stored in the on-chip ROMs, while the actual process is fulfilled using the Xilinx Floating Point (FP) IP cores [45], involving an integer-to-FP converter, an FP subtractor, and an FP divider. Consequently, the standardization process can be performed in T(*Stand.*) = T(*conv.*) + T(*Sub.*) + T(*Div.*) cycles, where T(x) depends on the actual configuration of the IP cores.

- **Classifier:**The classifier module implements an ANN model due to its better scalability and flexibility than a DT model. The ANN is designed with one hidden layer, whose number of neurons are determined through the evaluation in the following Section. The *sigmoid* function is used as the activation function. In particular, two layers of computation are required in our implementation, i.e., an input-hidden layer and a hidden-output layer, while the output of an arbitrary neuron at each layer involves the sigmoid result of the accumulation and the dot multiplication of its input and corresponding weights as follows:

$$O_i^{(j)} = sigmoid(\sum_{i=1}^{N} W_i^{(j)} \cdot I) \tag{4.14}$$

where I is the input vector to the i_{th} neuron at j_{th} layer, N is the input vector size, O_i and W_i are the corresponding output vector and weights of the neuron. The final classification result is, thus, based on the maximum pooling of the output layer.

The cardinal design of an ANN is the implementation of a neuron, which primarily consists of (1) memory storage that maintains weights and biases of layers and intermediate results, (2) computational logic that fulfills the equation 4.14, and (3) the class prediction based on the max-pooling. ROMs are employed to store the pre-computed weights and bias for each layer while a RAM is employed to store the intermediate output values of the hidden layer which serve as the inputs of the output layer. In order to implement the aforementioned dot multiplication and the accumulation efficiently, we take advantage of the Fused Multiply-Add (FMA) mode of the Xilinx FP IP core with a feedback logic. Furthermore, to simplify the *sigmoid* function design, we employ a piecewise linear approximation of the original function whose maximum absolute error of approximation is 0.005 [20]. To further reduce the design overhead of the ANN, the *sigmoid* function in the output layer is excluded without affecting the class prediction due to the monotonicity of the *sigmoid* function. Accordingly, the entire calculation in a single neuron for a N-length input vector takes T(*neuron*) = N $\times$ T(*mul-add*) [+ T(*sigmoid*)] cycles to finish.

Although a fully-parallel design of the ANN can produce data with optimal timing, the implementation overhead is overwhelming and is proportional to the number of neurons in the ANN structure, thus, may not be affordable. In contrast, we employ a serial design, which is optimized for minimal design overhead. As a result, our ANN consist of one instance of the neuron, while the latency to finishing the entire classification takes T(*classify*) = (H + C) $\times$ T(*neuron*), where H is the number of neurons at the hidden layer and C is the number of program classes.

- **Anomaly Detector:** To take the on-chip resource restriction for hardware design into account, a hardware-friendly extension of the multi-class classifier for anomaly detection is employed rather than implementing a separate detection module. Specifically, the *conjecture* of the anomaly detection is that the maximum probability of the

prediction in the ANN for a seen, i.e., legitimate, class is consistently larger (higher confidence level) than the maximum probability of the (mis)prediction for an unseen, i.e., suspicious, class (lower confidence level). Hence, a threshold can be studied for each legitimate class, while the workload identification is extended with the capability of anomaly detection as follows:

$$class = \begin{cases} suspicious, & if\ max.\ prob.\ <\ th(i) \\ argmax, & \text{otherwise} \end{cases} \tag{4.15}$$

By this means, the anomaly detection and classification tasks are integrated into one single implementation of the machine learning algorithm, which minimize the design overhead of the analysis module.

4.3.3 Experimental Results

The effectiveness of the proposed method in correctly identifying workloads and filtering suspicious workloads is assessed in this section. The classification results are illustrated using both spatial features and temporal features, while the optimal partition number `P` and the minimal required length of a branch address sequence are reported, which balance the effectiveness and design overhead.

The experiments were performed on a Linaro Linux host running Linux kernel 4.6, which is loaded on the Zedboard, a Xilinx Zynq-7000 series FPGA who embeds an ARM processor and ARM CoreSight Module. We collected the data trace generated through the CoreSight module directly from the hardware, decoded the package and performed the feature extraction mechanism in software for evaluation. Both common Linux commands and the MiBench [24] benchmark suite were used as our workloads, leading to the evaluation of 25 program families, while each family is executed with different arguments, creating multiple variations for classification in order to boost the resilience of our framework. In total, we

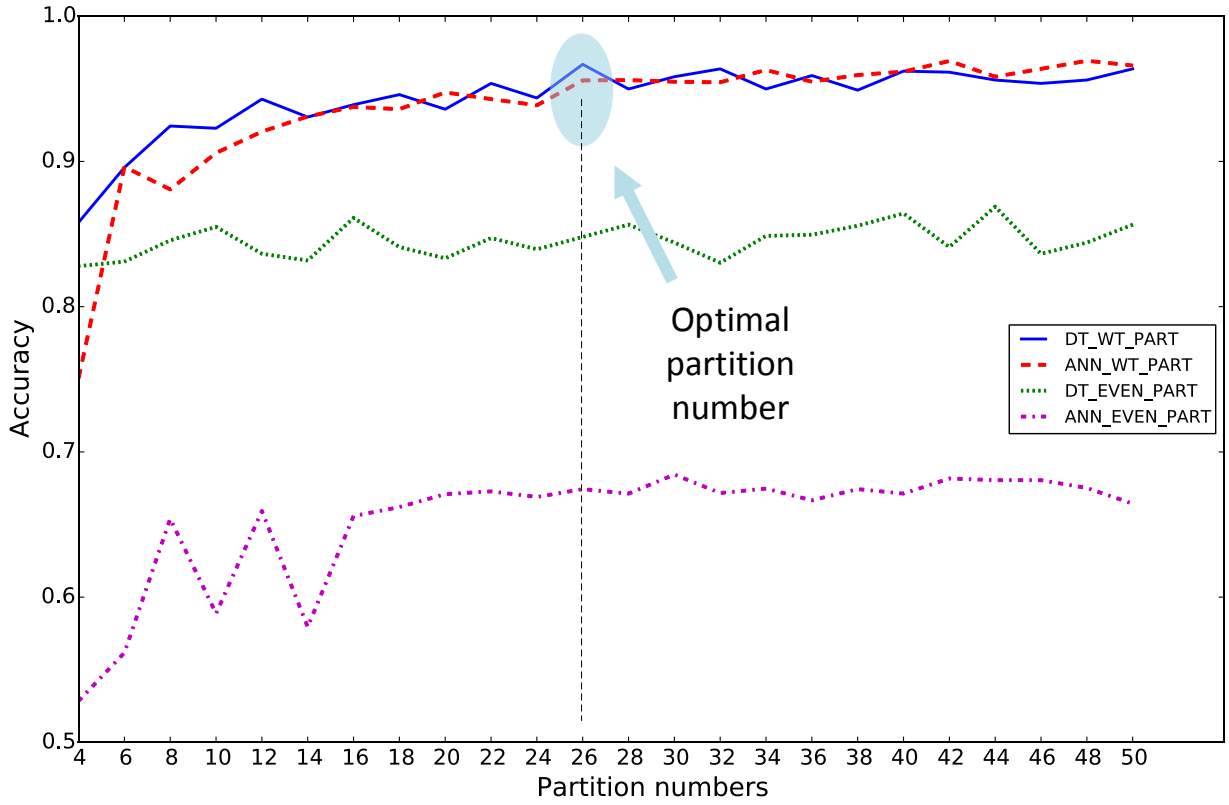
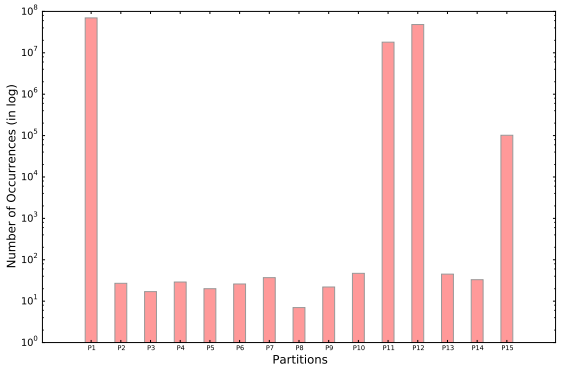


Figure 4.21: Average workload identification accuracy according to different partitioning method and partition number choice

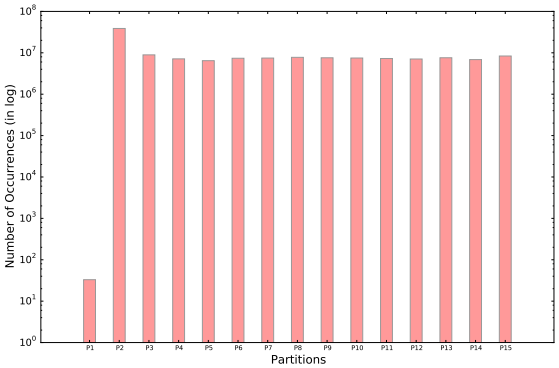
collect approximately 400 variations for each program family, which were split randomly in half for training and testing.

Partition Number P

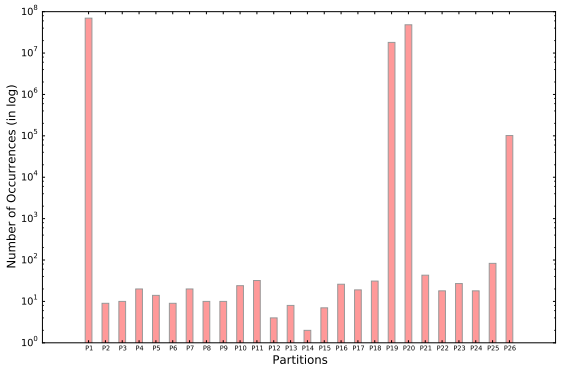
We first evaluate the impact on the effectiveness of the workload identification of our partitioning methodology and different choices of partition numbers. Both the evenly partitioning method and the heuristic weighted partitioning method was evaluated while the former was considered as a baseline design to compare with. We examined possible partition number ranging from 4 to 50, where the interval was 2. Figure 4.21 summarized the average identification accuracy over all the program classes, using both DT and ANN, corresponding to



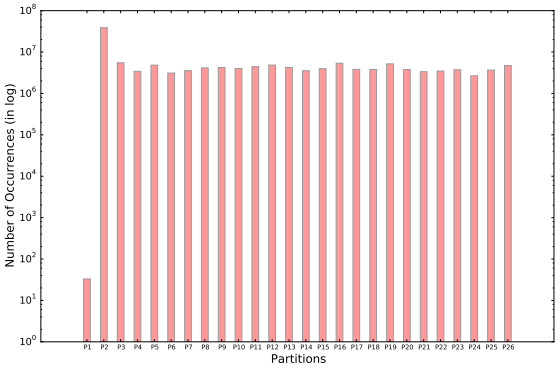
(a) Evenly partitioning, P = 15



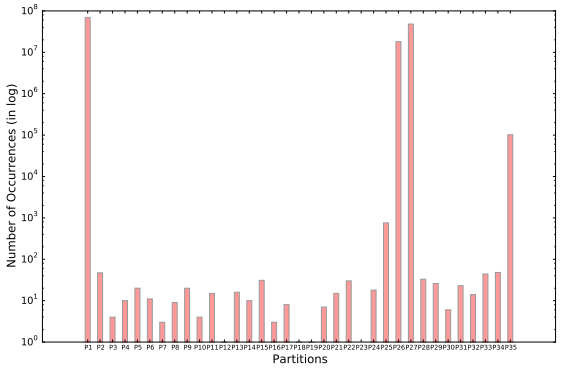
(b) Weighted partitioning, P = 15



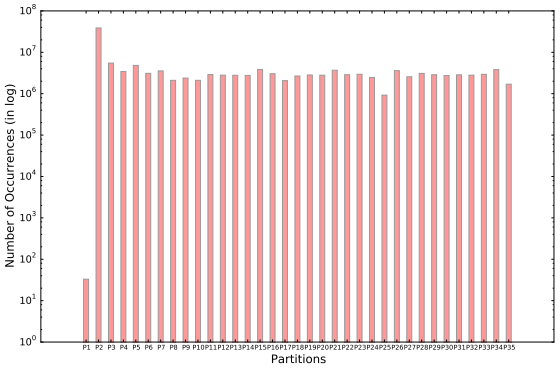
(c) Evenly partitioning, P = 26



(d) Weighted partitioning, P = 26



(e) Evenly partitioning, P = 35



(f) Weighted partitioning, P = 35

Figure 4.22: Distribution of counts of occurrence according to evenly partitioning vs. weighted partitioning in different partition size

different partitioning solutions. As may be observed, the evenly partitioning method led to no favorable result (the green and the maple line in Figure 4.21) in workload identification,

which reached an identification accuracy of approximately 83% for DT and 65% for ANN. Furthermore, the identification accuracy in this scenario does not change significantly with the increase in the partition number, which implies that the finer-grain granularity in the evenly partitioning does not produce a deeper view of program execution. A detailed distribution of counts of occurrence (in *log*) for multiple partition number choice in the evenly partitioning was illustrated in Figure 4.22a, 4.22c, and 4.22e. As may be observed, such the distribution remains extremely biased while more partitions are evaluated. Indeed, with evenly partitioning, most newly-added partitions are assigned to the insignificant area since every portion in the address space is treated equally. As a result, more partitions involved in the evenly partitioning contribute less in workload identification.

On the other hand, the DT and ANN perform well with the heuristic weighted partitioning method in workload identification. As shown in Figure 4.21, the average identification accuracy in both cases (the red and the blue line) increases monotonically according to the partitioning granularity. In particular, the DT obtained an approximately 10% gain in the average identification accuracy with finer-grained weighted partitioning, while the ANN obtained an approximately 20% gain in the accuracy. Compared with the evenly partitioning scenario, the DT ultimately surpassed the baseline with approximately 13% in performance, while the ANN surpassed the baseline with approximately 30% in performance. This observation implies that the weighted partitioning algorithm is able to break those biased areas, from which more significant information can be revealed. A detailed distribution of counts of occurrence (in *log*) for the weighted partitioning case was illustrated in Figure 4.22b, 4.22d, and 4.22f. Compared with their counterparts in the evenly partitioning scenario, the distribution is closer to a uniform distribution, which indicates that the significant area in an address space is broken further as the partition number increases. By this means, more partitions can convey more information, and, thereby, improves the distinguishability of program behavior.

Moreover, the identification accuracy in the weighted partitioning scenario is observed to reach a stability after a *'inflection point'*, while the increase in the partitioning granularity no longer has significant impact on the identification efficacy. This may be explained by the fact that the significance of the different portion of the address space has been well balanced, and thus, more partitions cannot bring additional information to distinguish program behavior. Therefore, we select the *'inflection point'* – in this case, 26 – as the optimal partition number, which balances the feature space size and the effectiveness of the classifier. Accordingly, we achieved an average identification accuracy of 96.68% and 95.57% for DT and ANN respectively.

Length of Branch Address Sequence

Assuming the optimal partition number being used, we next evaluate how the different length of the branch address sequence under evaluation influences the workload identification result. We evaluated the length of the branch address sequence varying from 1000 to 50000, where the interval was 1000. The identification accuracy, using both DT and ANN, according to different lengths of the sequence under evaluation was illustrated in Figure 4.23. As may be observed, workloads may not be distinguishable at the initial stage of their execution, since, generally, workload execution starts with some common initialization process. Along with the increase in sequence length under evaluation, however, the identification accuracy steadily rises. Similar to the partition number case, herein, we notice a *'inflection point'* as well, after which the workload identification accuracy stays stable, without affected by the sequence length under evaluation. As a result, we select 42000 as the optimal length of the branch address sequence when performing identification, in order to enable the real-time identification and maintain the balance between the response time and the effectiveness of the classifiers. A deeper view of the capability of our real-time identification is illustrated in Figure 4.24. Specifically, we report the percentage of the optimal length within the average

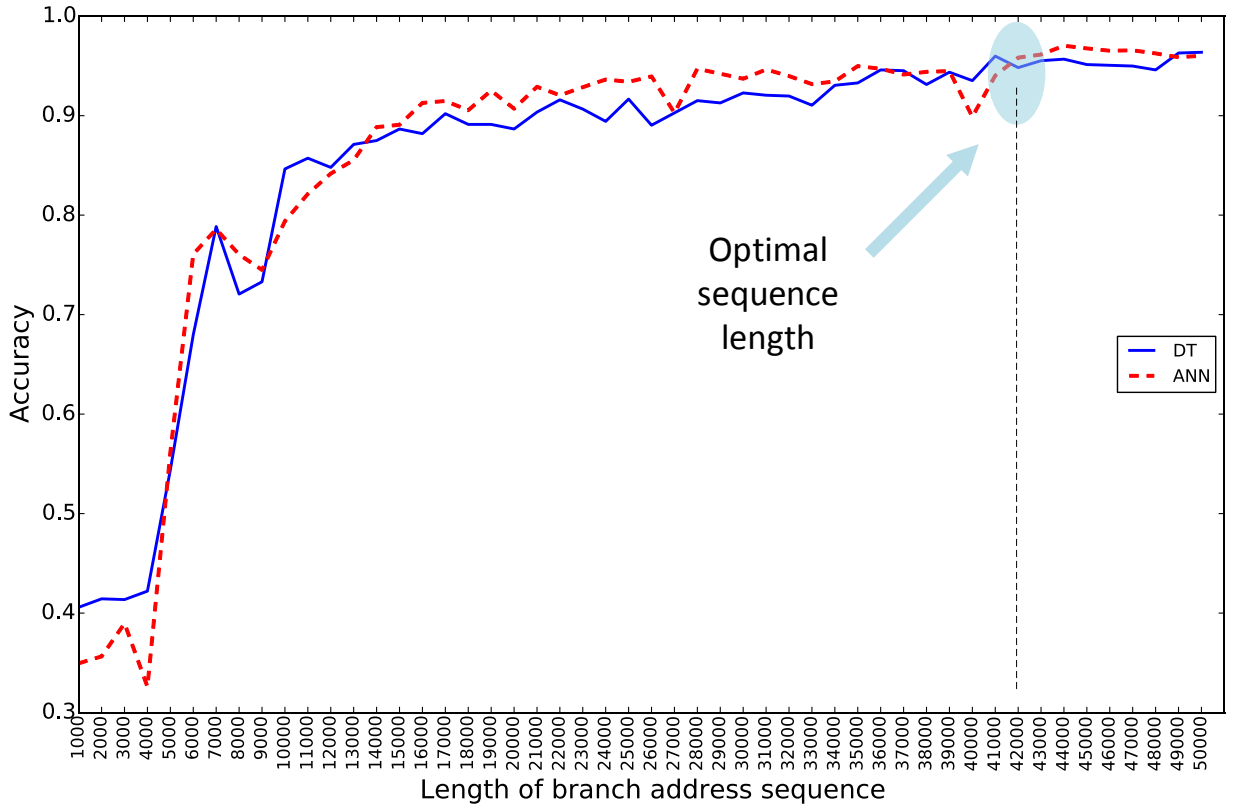


Figure 4.23: The average workload identification accuracy according to different sequence length

length of the original branch address sequences for each program class. As may be observed, we are able to identify workloads using 49.21% of their complete branch address sequence on average, while the best case is 20.9% and the worst case is 89.76%. Accordingly, with both partition number and sequence length under evaluation optimized, an average identification accuracy of 95.52% and 96.37% can be reached for DT and ANN respectively.

Using temporal features

Finally, we evaluate the effectiveness of our method using temporal features. The experiments were performed based on the optimization derived from the analysis above, while the performance of the counterparts using spatial features was considered as the benchmark

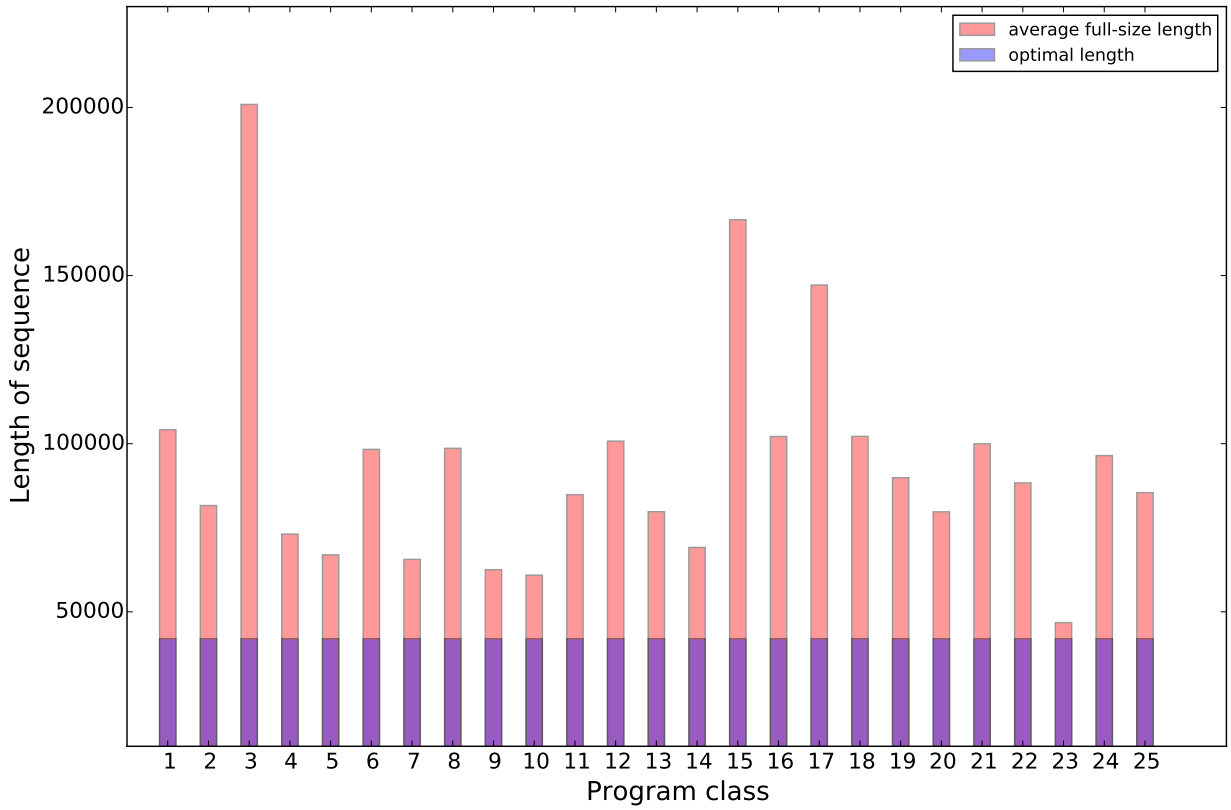


Figure 4.24: The optimal sequence length proportional to the full-size sequence length

performance. We first evaluate the effectiveness of the 2-gram model. Given the optimal partition number and the length of the branch address sequence, both DT and ANN achieved similar results as the benchmark, which is 95.45% for DT and 96.59% for ANN. However, the size of the feature space is squared, introducing a dramatic increase in design overhead. We also explored the feasibility of using the 2-gram model with the same size of feature space, i.e., we experimented with a choice of 5 partitions, which results in 25 features in total. Unfortunately, an average identification accuracy of 90.23% and 91.98% was achieved for DT and ANN, respectively, which does not surpass or reach the similar level of the benchmark performance. Nevertheless, compared with the identification result upon the spatial features with 5 partitions, an approximately 3% gain in identification accuracy was obtained. Table 4.5 summarized the comparison.

Table 4.5: Effectiveness of 2-gram model vs. spatial features

spatial features 26 partitions		**spatial features 5 partitions**	
DT	**ANN**	**DT**	**ANN**
95.52%	96.37%	88.58%	87.62%
2-gram 26 partitions		**2-gram 5 partitions**	
DT	**ANN**	**DT**	**ANN**
95.45%	96.59%	90.23%	91.98%

Table 4.6: Effectiveness of partition sequence vs. spatial features

spatial features optimal setting			
DT		**ANN**	
95.52%		96.37%	
spatial features len. 1000		**p. seq. len. 1000**	**p. seq. len. 1000 (no repeat)**
DT	**ANN**	**LSTM-RNN**	**LSTM-RNN**
40.58%	34.95%	35.14%	44.98%

The partition sequence feature is evaluated next. Similarly, we use the optimal partition setting in this experiment, i.e., 26 weighted partitions. Due to the limit in the computation complexity, the maximum sequence length that can be fed to our LSTM-RNN model is 1000, rather than 42000. Accordingly, an average identification accuracy of 35.14% was achieved under such setting, which is similar to the result using spatial features with the branch address sequence of length 1000, as shown in Figure 4.23. Apparently, The length of the partition sequence under evaluation limits the capability of the classifier to distinguish different workloads.

Nevertheless, a deeper view of the partition sequence reveals that the partition sequence consists of repeated patterns (e.g., prolonged repeated access in the same partition), which may be another source of ambiguity. Hence, we experimented with a variant of the original partition sequence feature, which maintained the same length but eliminated the repeated pattern. This enables capturing more temporal information in longer partition sequence

within a 1000-length window. As a result, an average identification accuracy of 44.98% was reached, which achieved approximately 10% improvement in the accuracy and surpassed the identification accuracy under scenarios of using spatial features as well as original partition sequence with the same length. Table 4.6 summarized the comparison. Consequently, we conclude that the temporal features are able to carry additional information to assist in distinguishing program behavior in certain scenarios, yet, with the cost of a dramatic increase in the design overhead, which limits their practicality. On the other hand, the spatial features remain the dominant factor in general in identifying different workloads.

To summarize our experimental results, the effectiveness of the workload identification based on spatial features is advantageous to the mechanism using temporal features, considering the trade-off between the identification accuracy and the design complexity. Through experiments, we select 26 as the optimal partition number, while the length of the branch address sequence is selected to be 42000 in order to enable real-time identification. As a result, our workload forensics framework is implemented based on the spatial features with the optimal setting as well as the ANN model.

Anomaly Detection

The effectiveness of the extension for anomaly detection was evaluated through experiments that selected arbitrary legitimate program classes as suspicious. The multi-class classifier was then trained with the remaining classes only while the unknown classes were included in the testing set only. The configuration of the machine learning algorithm (i.e., the features, partition size, sequence length, etc.) corresponds to the optimal setting concluded in the experiments for workload identification. Figure 4.25 illustrates the false negative (i.e., suspicious process classified as legitimate) rate as well as the false positive (i.e., legitimate process classified as suspicious) rate of identifying suspicious process classes according to different threshold settings. As may be observed, the hardware-friendly extension performed fairly

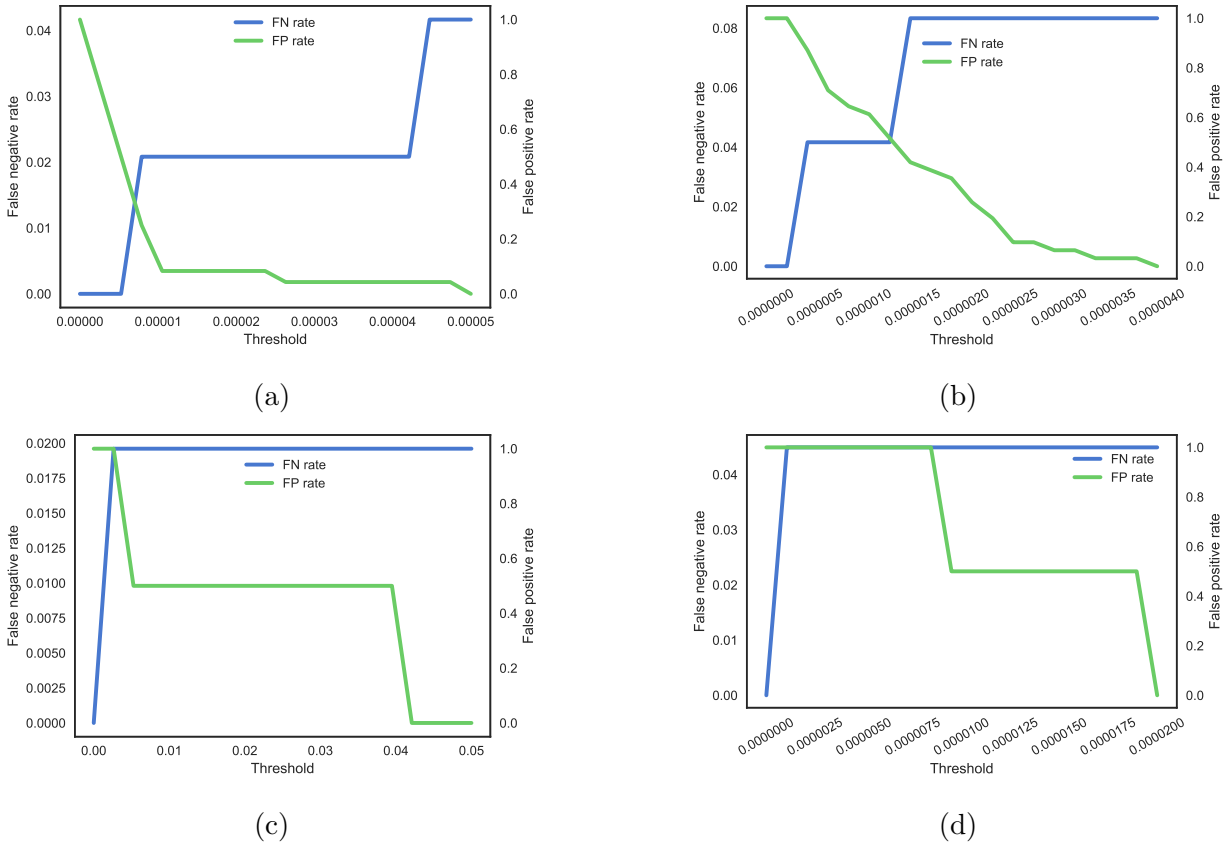


Figure 4.25: False negative rate vs. false positive rate according to the threshold for different program classes (subset)

well in filtering suspicious programs, reach an average FN rate of 4.5% and FP rate of 2% respectively, which confirms our conjecture. We note that although more advanced anomaly detection algorithms may potentially improve the results, significant design overhead may be introduced. On the other hand, the incurred overhead of our current solution, which extends the original design with the threshold comparison, is negligible. Nevertheless, the trade-off can be balanced in a different favor according to the specification and the available resources.

Table 4.7: Summary of effectiveness of hardware design

Classification accuracy		sigmoid at output layer	
		inclusion	**exclusion**
FP width	**32-bit**	56.12%	96.37%
	16-bit	56.12%	96.37%

Hardware Design Evaluation

In this Section, we evaluate the effectiveness and the design overhead of the hardware design of the proposed framework, according to the optimal configuration derived from the simulation results. The framework was implemented on Zedboard and was integrated with an ARM Cortex-A9 core. The processor operated at 333 MHz, while the ARM CoreSight Core and our framework were configured to operate at 100 MHz. The ANN is configured with 26 input features and one hidden layer with 10 neurons, which provides the best identification performance with minimal implementation overhead in our experiments. Two different instances, which employ the IEEE 754 half precision FP (16-bit) as well as the single precision FP (32-bit), were developed to evaluate the impact of the FP precision on the effectiveness of the framework. Furthermore, the inclusion and the exclusion of the *sigmoid* function at the output layer were evaluated as well to elaborate the impact of the function approximation on the effectiveness. Hence, four different implementations were evaluated accordingly.

The classification accuracy of the four implementations of the proposed framework is shown in Table 4.7, respectively. The best-case result matched the results obtained in the software simulation, which corroborates the effectiveness of our design. On the other hand, as may be observed, the selection of different FP precision has no impact on the effectiveness while the inclusion of the *sigmoid* at the output layer dramatically decrease the classification accuracy for both precisions. This can be explained by the use of the max-pooling mechanism by an ANN-based classifier in predicting. In particular, given arbitrary input vectors, an

Table 4.8: Summary of design overhead

	LUT util(%)	**BRAM util(%)**	**DSP util(%)**	**IO util(%)**	**Power (W)**
Proce-ssor	12.67%	2.14%	4.09%	32.5%	2.072
Frame-work	1.84%	1.79%	0.91%	2.5%	0.044

ANN generates its prediction based on the arguments of the maxima, or *arg max*, rather than the absolute values. As a result, as long as the ordering in the output vectors is retained, the slight error introduced by different FP precision can be ignored. On the other hand, the approximation of the *sigmoid* function applied in our design corrupts the original ordering (e.g., "A is greater than B" is approximated to "A equals B" when both A and B are larger or smaller than a threshold), and thus, leads to erroneous results in prediction. In a nutshell, it is observed that the exclusion of the approximated *sigmoid* function is necessary while the FP precision is insignificant, leading to a final design with half precision FP associated with the exclusion of the *sigmoid* at the output layer.

We evaluate the design overhead of the proposed framework with the implementation derived in the section above in two aspects as follows: (1) the area and power overhead introduced by our framework compared with an ARM processor and, (2) The estimated average latency, which measures the timing from the start of the workload execution to the point when framework outputs the identification result.

As shown in Table 4.8, the entire framework introduced additional use of 1.84% LUTs and 0.91% DSP, most of which are contributed by the FP arithmetic components. The additional BRAM utilization, on the other hand, is contributed by the neural network weights and bias ROMs. Moreover, an additional 2% overhead is introduced in the power consumption. Whereas our framework is non-intrusive to the processor execution flow, there is a delay between the start of a program execution and the identification outcome. Such latency depends on the average branch frequency `BF` (in *percents*) in a program profile. As a result,

the average latency to identify a workload will be T(*identify*) = T(*feat. gen.*) + T(*Stand.*) + T(*classify*) = 42000 ÷ BF × CPI + 234 + 2250 cycles (calculated by the equations defined in Section 4.3.2. Assuming the average BF to be 15% (according to the statistics in [24]) and CPI to be 1 to simplify the calculation, the proposed framework takes 865.68 μs to identify a workload at a 333 MHz processor clock with a 100 MHz framework clock.

4.3.4 Conclusion

In this work, a hardware-based on-line workload forensics framework was proposed, facilitated by the CoreSight module. We extensively explored the potential features that can be extracted from the trace generated by CoreSight module, upon which several machine learning models were evaluated. The proposed method was experimented in a Linux OS loaded on Zedboard, which embeds an ARM architecture. The parameters used for feature generation (i.e., the partition number and the sequence length) were optimized for real-time workload identification, leading to an average accuracy of 96.37% in classifying various program classes. The hardware implementation was evaluated on the Zedboard FPGA, integrated with an ARM processor, which incurred insignificant design overhead and identification latency.

CHAPTER 5

HARDWARE-BASED ROOTKIT DETECTION[1,2,3]

5.1 Overview

Alongside the computer forensics, malware detection is another branch to ensure system security, which has been widely studied and employed as well. Although numerous malware detection mechanisms have been developed, whose preliminary experiments have shown favorable results, there remains one species of malware, i.e., kernel rootkits, whose detection is far from promising. In general, kernel rootkits have unrestricted access to operating system (OS) resources due to their privileged implementation, and attempt to tamper kernel objects and inject malicious code stealthily. In this chapter, the feasibility of performing rootkit detection through hardware-based detection mechanism is explored. More Specifically, the proposed method aims at kernel rootkits which are assumed to (i) have full access to the OS memory image, and (ii) be able to make arbitrary modifications and execute malicious code in OS kernel space. As a result, the rootkits are able to hijack the control flow of arbitrary kernel services, e.g., system calls, and hook their malicious activities onto random benign processes. Furthermore, unlike previous malware or rootkit detection methods which require availability of known malware/rootkit samples during their program behavior modeling phase [43, 19, 37], we assume no prior knowledge of the kernel rootkits, i.e., the contaminated objects, the rootkit payload, or the binary image of the rootkits. In other

[1] © 2016 IEEE. Reprinted/portions adapted, with permission, from Liwei Zhou and Yiorgos Makris, "Hardware-based workload forensics and malware detection in microprocessors," in 17th International Workshop on Microprocessor and SOC Test and Verification (MTV), December 2016, pp. 45-50.

[2] © 2017 IEEE. Reprinted/portions adapted, with permission, from Liwei Zhou and Yiorgos Makris, "Hardware-based on-line intrusion detection via system call routine fingerprinting," in Design, Automation and Test in Europe Conference and Exhibition (DATE), March 2017, pp. 1546-1551.

[3] © 2018 IEEE. Reprinted/portions adapted, with permission, from Liwei Zhou and Yiorgos Makris, "Hardware-assisted rootkit detection via on-line statistical fingerprinting of process execution," in Design, Automation and Test in Europe Conference and Exhibition (DATE), March 2018, pp. 1580-1585.

words, we assume a *zero-day attack* scenario. Accordingly, two incarnations are proposed herein, i.e., static rootkit detection and dynamic rootkit detection.

5.2 Static Rootkit Detection

5.2.1 Threat Model

The hardware-based static rootkit detection ensures the integrity of an executed system call service routine during runtime in a way that is immune to tampering by software, hence, follows the on-line system architecture [48]. The idea is motivated by the fact that most malware detection methods rely on system call-related information, yet their logging/monitoring mechanisms rarely inspect the actual system call execution flow, which leaves room for malwares to evade detection through *system call hijacking*. System call hijacking enables an attacker to control the execution flow of one or several system calls; thereby, malicious code can be introduced and executed without the knowledge of the legitimate system user. In Linux OS, for example, this can be achieved through a kernel rootkit exploiting the Loadable Kernel Module (LKM), which contains user defined code and is intended to extend or customize the functionality of the original kernel. Compounding the problem, when the extended or customized functionality is no longer required, the kernel module can be unloaded, restoring the kernel to its original state and, thereby, leaving no trace.

In general, three types of system call hijacking attack can be launched, as shown in Figure 5.1:

- **System Call Table Redirection:** The simplest attack using the LKM is a redirection of the system call table. When a system call is invoked, instead of querying the original system call table to access the corresponding service routine, the OS is redirected to a different table whose content is controlled by the attacker and whose existence is unknown to the OS and the legitimate users.

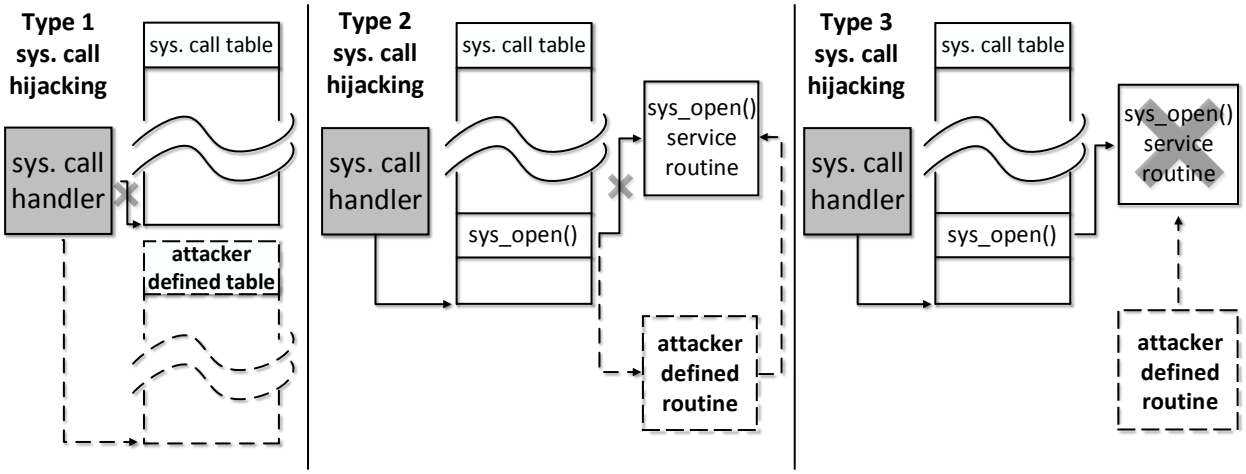


Figure 5.1: Three types of system call hijacking

- **System Call Table Modification:** Another attack option is to modify the value of certain entries in the original system call table rather than redirecting to a different table, as the latter may leave more of a trace and can be easily detected. In this way, the attacker can redirect the service routine of certain system calls to his/her own malicious code snippet. After the malicious code finishes, control is returned to the original service routine so that the OS remains oblivious to the attack.

- **Service Routine Modification:** A more complicated option is to directly modify the service routines of one or more system call. In this case, the system call table as well as its entries remain unmodified, yet the actual service routines are contaminated with additional/different instructions. A smart attack may still operate covertly and hide its presence from the OS by incorporating the general service provided by the original routine.

Theoretically, the type 1 attack requires the least design complexity but suffers more risk to be detected, while the type 3 attack is the most complicated by design yet the most likely to evade detection. Armed with the capability of system call hijacking, an attacker can easily spoof OS-level and hypervisor-level intrusion detection methods. Indeed, upon invocation of

a system call, these methods typically log and validate the system call ID or series of IDs, yet have no mechanism of attesting that the legitimate service routine is executed.

Accordingly, the proposed hardware-based rootkit detection system addresses the three types of system call hijacking separately by following an on-line system architecture, which consists of two main components, i.e., the logging component and the validation component. In particular, the logging component collects three critical pieces of information related to the integrity of system call execution, namely the base address of the system call table, the contents of the system call table, and the actual system call service routines. When a system call is invoked, the validation component, then, contrasts these information against their corresponding valid signatures in order to detect the three types of system call hijacking attacks.

5.2.2 System Design

The proposed hardware-based intrusion detection system, as Figure 5.2 shows, consists of two main components: a *data logging component* and a *validation component*. The data logging component collects three critical pieces of information related to the integrity of system call execution, namely the base address of the system call table, the contents of the system call table, and the actual system call service routines. Using this information, the validation component seeks to detect the three types of system call hijacking attacks included in our threat model. The base address of the system call table, as well as its contents, are retrieved through the Basic Input/Output System (BIOS), when the OS kernel is booted. Therefore, we also store this information directly in the hardware and contrast it against the real-time values every time a system call invocation occurs, so that any unexpected modification can be detected and an alarm can be raised to suspend execution.

Attesting validity of the actual system call routine, however, is slightly more involved. Specifically, our method employs custom hardware to generate a fingerprint for each basic

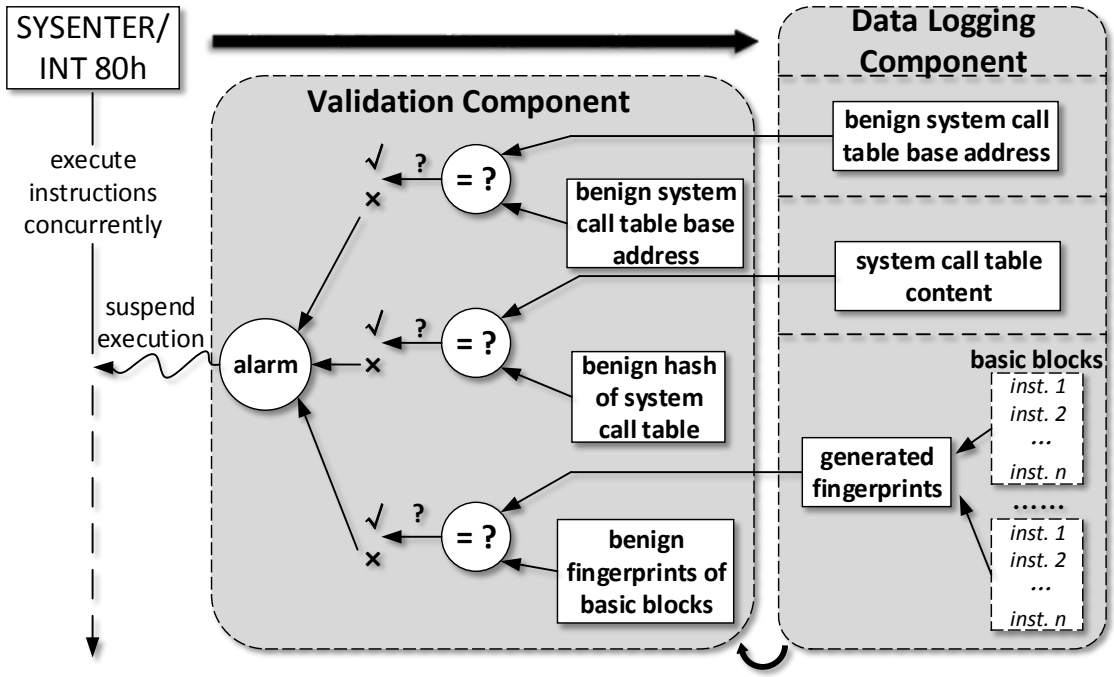


Figure 5.2: High-level design of proposed method

block of the executed routine and to compare it against a set of known acceptable fingerprints for this routine. The choice of a basic block, as opposed to the entire system call service routine, as the minimum entity to be fingerprinted is driven by practicality. A basic block is a snippet of atomic code executed between two control flow transfers. Therefore, the actual instructions executed are fixed, hence the golden fingerprint of a basic block can be statically computed. In contrast, the instructions executed by the entire system call service routine depend on the arguments with which it is invoked at run-time; hence there is a multitude of golden fingerprints which are not only harder to exhaustively identify but may also leave more room for malicious modifications to go undetected.

```
int 80h
lea esi,0[esi]
...
jae 0xc162d80a
call dword ptr 0xc16380a0[eax*4] /*system call table is referred to here*/
```

(a) INT 80H handler

```
sysenter
mov esp,dword ptr 0xffffde84[esp]
...
jae 0xc162d80a
call dword ptr 0xc16380a0[eax*4] /*system call table is referred to here*/
```

(b) SYSENTER handler

Figure 5.3: Two types of the system call handler in x86

5.2.3 Implementation

Data Logging Component

To extract system call related information directly from the hardware without assistance from upper-level resources, such as `system.map`, we exploit several x86 hardware conventions, as described below.

- **System call table address:** In a 32-bit x86 architecture, two methods can be used to invoke a system call. The first one uses the conventional software interrupt, through `INT 0x80`. In particular, this instruction consults the Interrupt Descriptor Table (IDT) to identify the entry point to the system call table, which is then indexed with a system call code to execute corresponding service routine. As shown in the first basic block of the `INT 0x80` handler in Figure 5.3a , the exact base address of the system call table, in this case `0xc16380a0`, can be obtained through decoding the last instruction. The second method for invoking a system call uses `SYSENTER` instructions. These instructions were introduced by Intel in order to enable fast entry to the kernel and

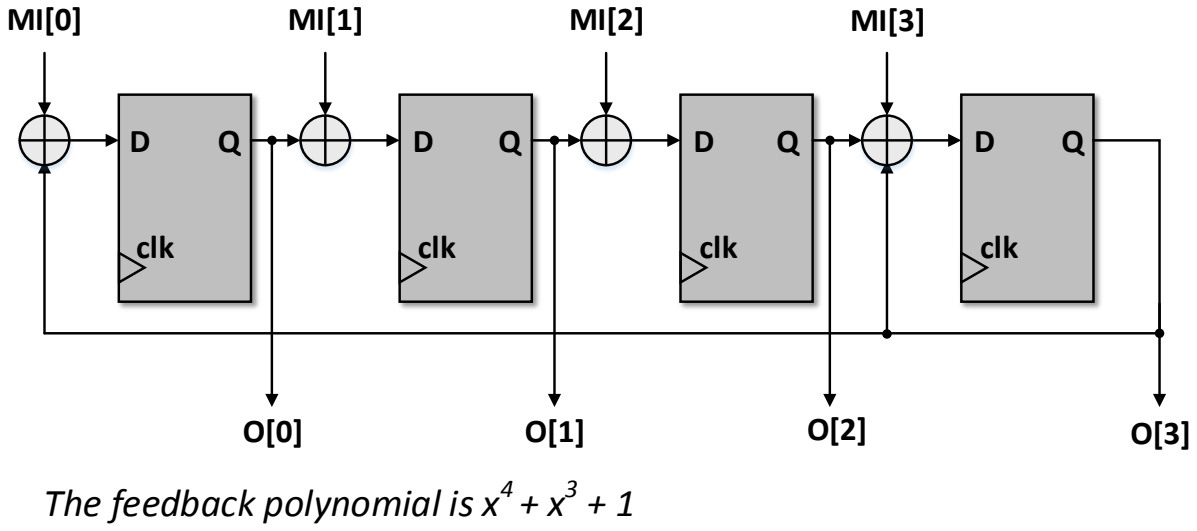


Figure 5.4: An example of a 4-bit MISR

avoid the overhead incurred by software interrupts. Similarly, the same base address is referred to in the first basic block of the `SYSENTER` handler, as shown in Figure 5.3b.

If an attacker launches a Type-1 attack to redirect access to the system call table, the base address of the attacker-defined table must appear as a target address in the first basic block of the system call handling routine. Therefore, comparing the actual address to the legitimate one at run-time directly in hardware helps in detecting such attacks.

- **System call table content:** In addition to the base address of the system call table, its content is also known. In this work we store a golden fingerprint, which is generated using the mechanism introduced in the following section, for the entire system call table. When a system call is invoked, the fingerprint of the employed table is computed and compared to the golden one in hardware, in order to detect Type-2 attacks.

- **System call service routine**: In order to detect Type-3 attacks, we need the ability to analyze instruction-level behavior of system calls. However, logging the entire

instruction flow would introduce unacceptable hardware overhead and is, generally, unnecessary. Instead, we employ a MISR to compress the instruction flow and generate simple fingerprints.

A MISR is a variant of a Linear Feedback Shift Register (LFSR). A standard LFSR is a shift register whose output is a linear function of its previous state, where the feedback input bit is generated by the XOR/XNOR function of a subset of the register bits. A MISR has the same structure, but additional input bits are fed through an XOR/XNOR gate to every flip-flop of the shift register in each cycle, as shown in Figure 5.4. As a result, the next state of the MISR depends on both the current state and the input bits.

Using a MISR for our purpose has three advantages: (1) the hardware structure of a MISR is relatively simple, involving only D-Flip-Flops (DFF) and XOR gates; thus, it incurs low design overhead. The number of DFFs and XORs is decided by the characteristic polynomial of the underlying LFSR. In our case, since we seek to compact instructions and the maximum length of a single instruction in x86 is 15 bytes, the degree of the characteristic polynomial has to be at least 120[4]. Specifically, we chose to implement a MISR using $x^{120} + x^{119} + 1$ as the characteristic polynomial. (2) A MISR is scalable and can process multiple inputs simultaneously, independent of the input length; this allows us to efficiently process entire instructions in order to generate fingerprints for basic blocks. (3) The MISR has a relatively low aliasing probability. Aliasing occurs when identical signatures are generated for different input sequences and can undermine our ability to detect invalid instruction sequences. An approximation of the aliasing probability of a MISR is 2^{-n}, where n is the degree of its characteristic polynomial. In our case, since n is 120, the aliasing probability is as low as 2^{-120}, which is negligible.

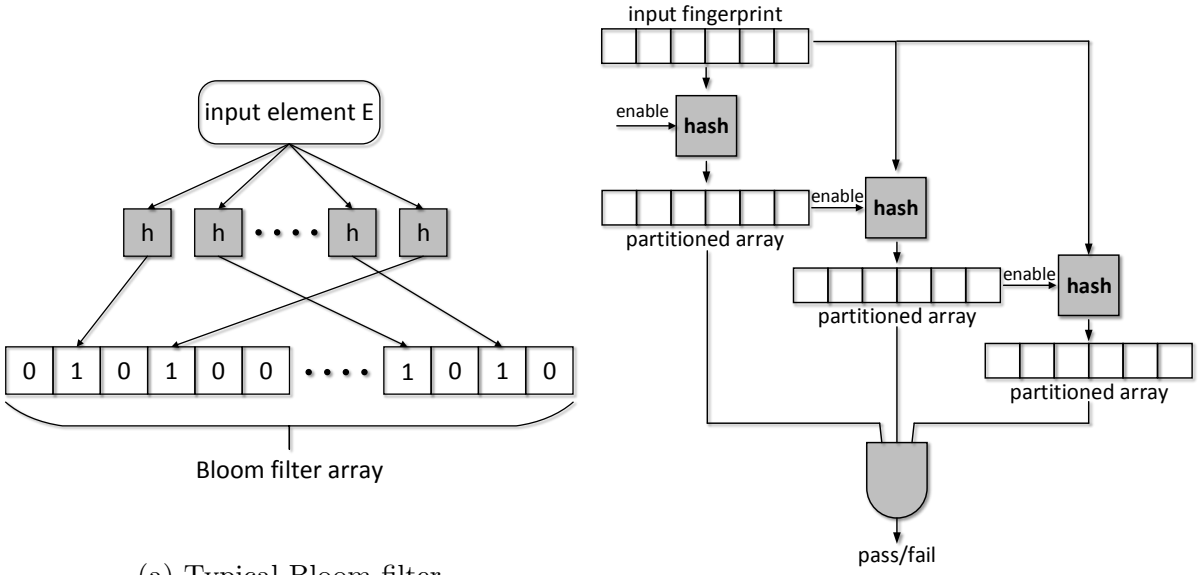
[4]Intel 64 and IA-32 Architectures Software Developer's Manual

In this work, execution flow of a system call service routine is described by multiple fingerprints, one for each of its basic blocks. The generated fingerprints are then fed into the on-line validation component, which we will describe next, to detect intrusion. To associate the generated fingerprints with the corresponding system call, we use the system call code as an identifier. In x86, the system call code is stored in the `EAX` register when an `INT 0x80` or `SYSENTER` instruction is executed, hence our hardware obtains it directly from there.

5.2.4 Validation Component

The role of the validation component is to examine the validity of the fingerprints generated by the logging component for the basic blocks of a system call service routine. In particular, the golden fingerprints for each system call are identified through static code analysis and programmed in the validation component, which then checks membership of an on-line generated fingerprint in the appropriate golden set. A failed membership test implies that the system call service routine is invalid. Considering the potential design overhead of the validation component, explicitly storing in hardware all golden fingerprints for each system calls and comparing them in parallel against an on-line generated fingerprint would be prohibitively expensive. Therefore, instead of using a lookup table or a hash table for this purpose, we employ a Bloom filter for compactly storing the golden fingerprints and rapidly performing membership tests.

A Bloom filter is a space-efficient probabilistic data structure, used to test whether an element is a member of a set [8]. As shown in Figure 5.5a, a typical Bloom filter consists of an m-bit array and implements k different hash functions h_i, $i \in [1, k]$, each of which maps an input element E to one of the positions in the array through a uniform random distribution. An empty Bloom filter is an array with all 0s. When adding an element, all k position bits mapped by the hash functions for this element are set to 1. As a result, an element is a member of the set if and only if $\forall i \in [1, k], h_i(E) = 1$.



(a) Typical Bloom filter

(b) Validation component

Figure 5.5: Implementation of the validation component using Bloom filter

A Bloom filter never misidentifies a valid member of the set as a non-member. However, it is possible that due to collision in the outputs of the hash functions, a non-member may be accepted as a member of the set, which in our case would imply that an invalid fingerprint may evade detection. Fortunately, appropriate choice of m and k can bound the probability of this evasion detection occurrence [3]. Furthermore, our design leverages a partitioned and fully-pipelined Bloom filter architecture. Partitioning, which splits the m-bit array into k sections that can be separately queried by the k hash functions, may further reduce detection evasion, since the outputs of the k hash functions themselves cannot suffer from collision [3]. Pipelining, on the other hand, allows us to disable the computation of the next hash function when the output of the previous hash function is 0, which indicates that the input element is certainly not a member of the set, thereby reducing power consumption.

In a Bloom filter, for a given array size of m bits and set cardinality n, the optimal number of hash functions k_{opt}, which minimizes detection evasion probability p, is $k_{opt} = \frac{m}{n} ln2$. For the optimal value k_{opt}, m becomes a linear function of n, which can be calculated as

$m = \left\lceil -\frac{nlnp}{(ln2)^2} \right\rceil$. In this work, the maximum n (i.e., basic blocks in a system call routine) does not exceed 3500, while p is set to 3%. As a result, the chosen values for m and k are 24576 bits (3 KB) and 6, respectively.

Our Bloom filter implementation uses the hardware-friendly hash function design suggested in [12]. Specifically, given a b-bit input element E, its hash is calculated as

$$h(E) = (S_1 \cdot E_1) \oplus (S_2 \cdot E_2) \oplus \cdots \oplus (S_b \cdot E_b)$$

where S_i is a random seed coefficient $\in [1, log_2 m)$ and the operation $\cdot$ is,

$$S_i \cdot E_i = \begin{cases} S_i, & if\ E_i = 1 \\ 0, & \text{otherwise} \end{cases}$$

Further optimization is applied to the construction of hash functions. A classic Bloom filter requires k independent instances of hash functions, which introduces significant overhead when k grows. However, as described in [28], two independent hash functions h_1 and h_2, referred to as primary hash functions, are sufficient, while the remaining $k-2$ hash functions can be obtained through $h'(x) = h_1(x) + j \cdot h_2(x)$, where j is an arbitrary integer. Thus, the design overhead can be approximately bound by the cost of two hash functions, regardless of the value of k. Detailed implementation of the validation component is shown in Figure 5.5b. Finally, the overall architecture of the proposed method, which interfaces the logging and validation components with the microprocessor pipeline, is illustrated in Figure 5.6.

5.2.5 Experimental Results

Evaluation of this hardware-based intrusion detection method was also performed in Simics with the same configuration as the one in the case of workload forensics. To evaluate effectiveness of this method, a static analysis was first performed to collect the golden fingerprints of the basic blocks for each system call service routine, and to program the corresponding Bloom filters. Next, MiBench was used to collect fingerprints for legitimate workload. On

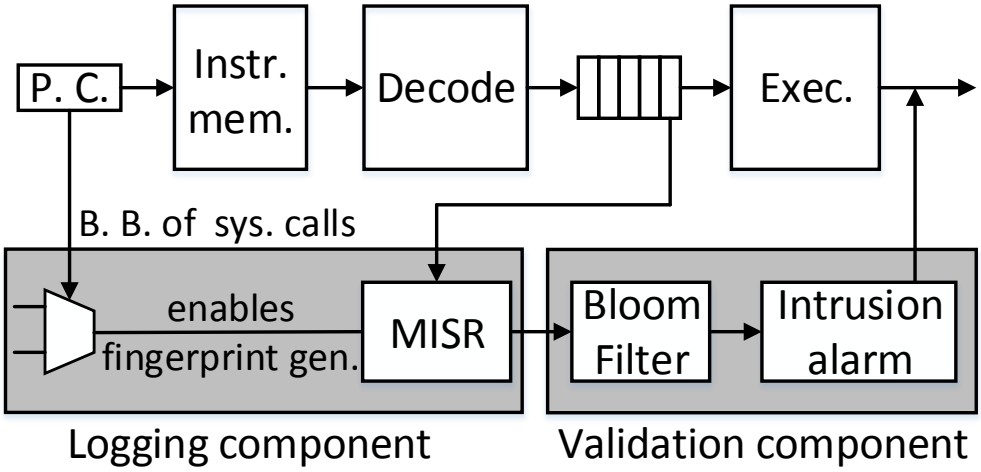


Figure 5.6: Architecture of proposed method

Table 5.1: Statistics for legitimate workload

system call	**invocations**	**fingerprints**
sys_open	1103	2708
sys_read	4480	3325
sys_write	5504	1883
sys_close	181	1470
sys_rt_sigprocmask	5640	146
sys_rt_sigaction	1029	392
sys_mmap2	218	1170
sys_ioctl	318	231
sys_brk	125	179

the other hand, to collect fingerprints for contaminated workload, five rootkits which exploit LKM to launch system call hijacking attacks of the three types introduced in Section 5.2.1 were evaluated. In order to hook our logging component onto the program counter and monitor the instruction flow, we exploit the *haps* feature of Simics. We note that, for reasons explained in Section 5.2.5–'Limitation', in our experiments we only fingerprint basic blocks entered through direct jump instructions.

Table 5.2: Kernel rootkit detection summary

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

Effectiveness

Table 5.1 summarizes the number of times each system call was invoked by the legitimate workload, as well as the corresponding number of distinct fingerprints (only the most frequently invoked system calls are shown). All of these fingerprints were processed by the corresponding Bloom filter and passed the validation process, i.e., no false alarms occurred.

Table 5.2 summarizes the five kernel rootkits which we used to launch system call hijacking attacks. Following the definitions in Section 5.2.1, the first one is Type 1, the next three are Type 2 and the last one is a Type 3 threat. All five were successfully detected by our method, as they invoked system calls whose service routine execution generated fingerprints that were rejected by the corresponding Bloom filter.

Design Overhead

Since the logging and validation components are implemented directly in hardware and do not interfere with the original microprocessor execution flow, the performance overhead of our system is expected to be zero. As a point of reference, similar work developed at the hypervisor level, introduced additional performance overhead of 262.3 ms [43].

To evaluate the hardware design overhead of our method, we use a predictive 45nm Process Design Kit (PDK) [41]. The logging component and the validation component are separately synthesized in Synopsis Design Vision. Since the library does not contain a memory cell, we report the memory cost of the bit array and the seed coefficients of the Bloom filter separately, in terms of bytes required.

Table 5.3: Design overhead summary

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

Table 5.3 summarizes the incurred overhead for each of the two components. Compared with a 45nm Intel Processor[5], whose area is 107 mm^2 and average power consumption is 65 W, the total overhead of this hardware implementation is negligible. Furthermore, while other hardware-based methods performing similar analysis introduce additional 2% area overhead in general [18, 14], our implementation consumes hundreds of times less in area. Regarding the seed coefficient lookup table, two tables are required for the two independent primary hash functions, in which 120 different random 12-bit values are programmed. Therefore, the memory cost of the seed coefficient lookup table is 360 B in total. Considering that the total system call number in Linux 2.6 is 336, the memory cost of the bit array for the Bloom filters is 0.98 MB.

Limitation

In modern microprocessors, control flow transfers may involve either direct or indirect jump instructions. In the former, the starting point of the next basic block is explicitly known, but in the latter it is computed at run-time and may redirect execution to an instruction that is not at the beginning of a statically identified basic block. Hence, indirect jumps limit our ability to retrieve all possible basic blocks within an execution flow. Therefore, in order to avoid false alarms, basic blocks reached via an indirect jump are not fingerprinted. However, similar to other hardware-based methods which rely on behavior mod-

[5] `http://ark.intel.com/products/35605`

eling [43, 19, 37, 47], if a non-zero false-positive rate is acceptable such fingerprinting can be enabled to opportunistically detect indirect jump-based attacks. Evidently, such attacks constitute a very challenging problem for hardware and software-based intrusion detection methods alike. Software solutions developed are vulnerable to tampering, incur relatively high implementation overhead which makes their deployment impractical [15, 7], and have limited effectiveness, as implied in [10]. These facts motivate our hardware-based dynamic rootkit detection proposed in the next section.

5.2.6 Conclusion

In this work, a low-cost hardware-based approach was introduced for performing on-line intrusion detection through system call routine fingerprinting. Unlike software-based methods, this hardware-based method extracts the required information directly in hardware, making it impervious to software attacks. Herein, an incarnation of this general idea was demonstrated, which logs and generates fingerprints using a MISR and examines their validity by checking their membership in a pre-specified set of golden fingerprints, implemented as a Bloom filter. The proposed method was evaluated on a 32-bit x86 architecture running Linux OS, implemented in Simics. Experimental results, using the Mibench suite as legitimate software and kernel rootkits launching system call hijacking attacks as malware, reveal that the fingerprints of the contaminated routines can be fully differentiated from those of the legitimate code. The hardware cost of our method in a 45nm process is 9229.72 μm^2 in area and 21.45 mW in power, as well as approximately 1 MB of memory, which is negligible compared to the size of a modern microprocessor. Nevertheless, the intrinsic static analysis may prevent this method from detection software tampering in dynamic program behavior, which can be launched through modifying code blocks accessed through indirect jump.

5.3 Dynamic Rootkit Detection

5.3.1 Threat Model

In this work, a threat model with a broader definition is considered. Particularly, the proposed method aims at kernel rootkits which are assumed to (i) have full access to the OS memory image, and (ii) be able to make arbitrary modifications and execute malicious code in OS kernel space. As a result, the rootkits are able to hijack the control flow of arbitrary kernel services, e.g., system calls, interrupt handler, etc., and hook their malicious activities onto random benign processes. Furthermore, unlike previous malware or rootkit detection methods which require availability of known malware/rootkit samples during their program behavior modeling phase [43, 19, 37], we assume no prior knowledge of the kernel rootkits, i.e., the contaminated objects, the rootkit payload, or the binary image of the rootkits. In other words, we assume a *zero-day attack* scenario.

5.3.2 System Design

Due to the limitation of static computation in the rootkit detection method proposed above, i.e., a smart attacker may be able to launch an attack through the execution path accessed via indirect jumps, and thus, evade the static detection, a dynamic rootkit detection method is proposed herein to address this limitation, which leverages behavior modeling of runtime program execution in order to distinguish rootkit-infected execution from the legitimated counterpart.

Traditional dynamic malware detection methods seek to model program behavior and, therefore, train a 2-class classifier, in order to distinguish malicious processes from benign ones. In other words, these methods detect malware through *inter-process behavior deviation*. However, such a detection mechanism may fail under rootkit attack scenarios. For example, an attack can be launched by implanting a rootkit that injects malicious code in the original

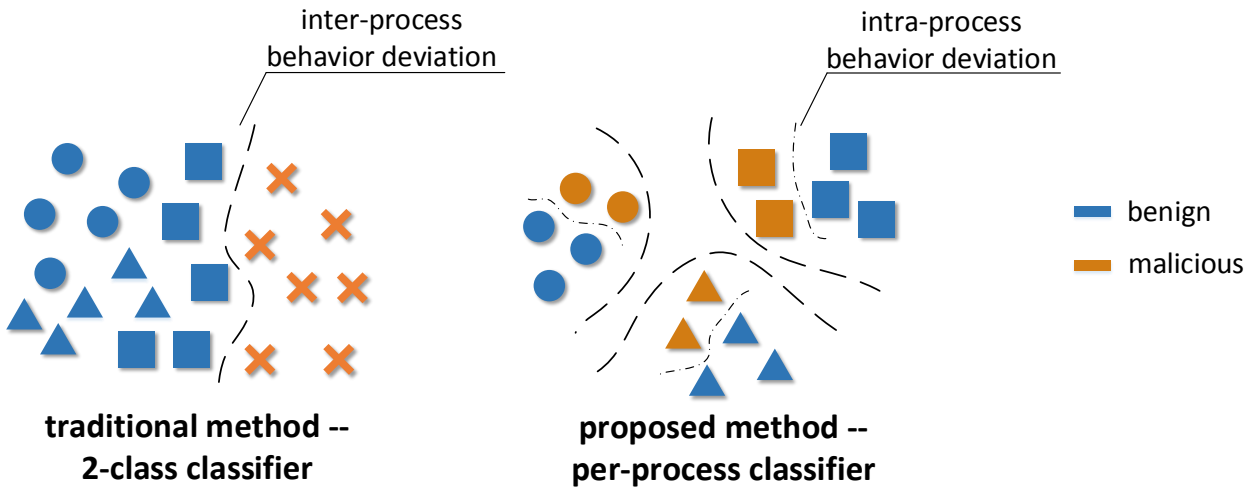


Figure 5.7: Traditional vs. proposed method

system call table. This results in distortion in the execution flow of existing processes, rather than creation of a new (covert) malware instance. In such cases, rootkit-infected processes, whose behavior deviates only slightly from their legitimate version due to malicious actions, may not be detected. In contrast, the proposed dynamic rootkit detection mechanism herein models program execution profile individually for each process, using a machine learning approach, in order to identify whether a process is rootkit-infected. In other words, this detection mechanism relies on *intra-process behavior deviation*, as shown in Figure 5.7. As a result, a more precise view of process execution profile is constructed at a finer granularity and, thus, even slight deviations of process execution flow incurred by kernel rootkits can be detected. In a nutshell, the dynamic rootkit detection method accords to the program-centric and behavior-based paradigm.

The actual implementation of this method follows a off-line system architecture, which mines the architecture state and extracts relevant information exclusively from the hardware, and off-loaded the collected data to a trusted software environment, which is isolated from the original OS, wherein the rootkit detection is performed by a machine learning entity which has been trained to model the intra-process behavior. The rootkit detection is performed

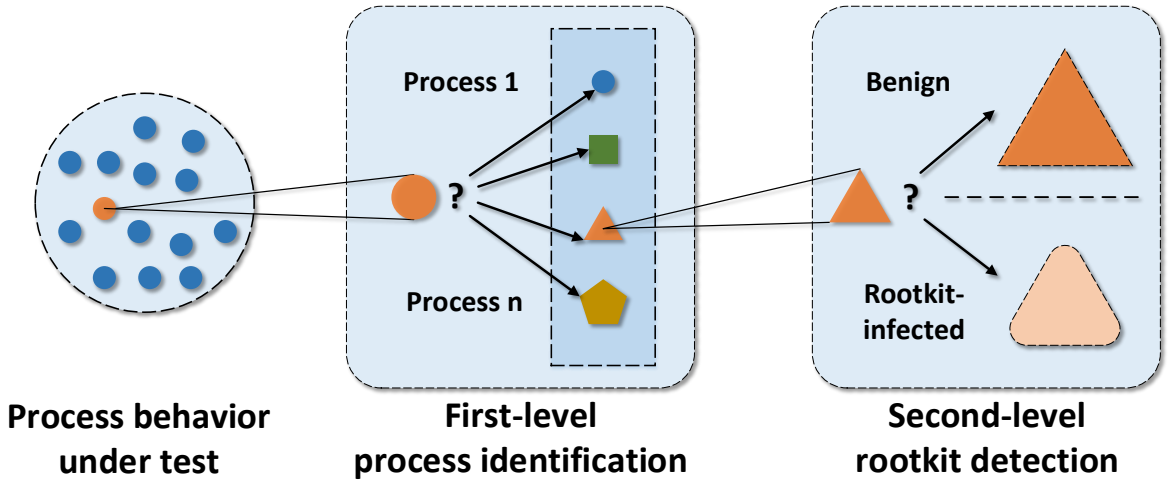


Figure 5.8: Rootkit detection flow

through a hierarchical mechanism, as shown in Figure 5.8. When a new process sample arrives, a first-level process identification is applied to identify what process class it belongs to. After that, a second-level rootkit detection is performed on the corresponding process class independently, in order to investigate whether a process sample is truly benign or rootkit-infected.

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

5.3.3 Implementation

Process Identifier and Behavior

This work aims at the x86 architecture and thus, similarly, the `CR3` value is used as the identifier of a process.

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

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

In x86, four general purpose registers, i.e., `eax`, `ebx`, `ecx`, and `edx`, are most frequently used. This method collects counts of the three types of dependencies on each of the registers as the data dependency-related features. Furthermore, instructions can operate in user mode

Table 5.4: Summary of feature set

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

or kernel mode in a modern OS. Data dependency statistics are collected separately for these two modes, leading to a deeper understanding of how a process operates in its user space and in kernel space. Ultimately, for each process (represented by `CR3` value as mentioned in Section 4.1.2), 24 data dependency-related features are collected.

Regarding branches in program execution flow, 3 types of branches are considered, including intra-user, user-kernel and intra-kernel branches. Intra-user branches involve jumps between user-space instructions, capturing the functionality of a program in user mode. User-kernel branches, on the other hand, involve transition between user and kernel mode. Such transition may occur due to either software interrupts, which are launched actively by program execution, or hardware interrupts, which are asynchronous with the program execution flow. Since this method aims at modeling program behavior with minimal impact on the underlying environment, only branches introduced by software interrupts, launched by programs explicitly through `SYSCALL` or `INT` instructions, are considered. Finally, similar to intra-user branches, intra-kernel branches are collected accordingly. Table 5.4 summarizes the features considered in this work.

Machine Learning for Rootkit detection

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

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

  KNN is a non-parametric classification algorithm which classifies samples based on spatial relationship in their feature space. It computes the k nearest neighbors of a sample using Euclidean distance and assigns the sample to a class based on majority voting among these neighbors. SVM, on the other hand, generates a hyperplane which separates the transformed feature space into labeled sub-spaces, while ensuring maximal separation among them. ANN exploits a multi-layer structure, where each layer consists of multiple nodes, i.e., *neurons*, which are interconnected with nodes in adjacent layers. Through stacked layers, ANN maps the original inputs, via an activation function on each neuron, to a final labeled space, accomplishing the classification. In our implementation, we used KNN from the Matlab library, SVM from the LIBSVM library [11] and ANN from Keras [13].

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

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

$$\tilde{f}(x) = \frac{1}{nh^d} \sum_{i=1}^{n} K(\frac{1}{h}(x - X_i)) \tag{5.1}$$

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

$$K_e(t) = \begin{cases} \frac{1}{2} c_d^{-1}(d+2)(1 - t^T t), & t^T t < 1 \\ 0, & otherwise \end{cases} \tag{5.2}$$

where $c_d^{-1} = 2\pi^{d/2}/(d \cdot \Gamma(d/2))$ is the volume of the unit d-dimensional sphere [42]. A rule-of-thumb choice of h is:

$$h = \left\{ 8c_d^{-1}(d+4)(2\sqrt{\pi})^d \right\}^{1/(d+4)} n^{-1/(d+4)} \tag{5.3}$$

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

$$\hat{f}(y) = \frac{1}{n} \sum_{i=1}^{m} \frac{1}{(h \cdot \lambda_i)^d} K(\frac{1}{h \cdot \lambda_i}(x - Y_i)) \tag{5.4}$$

where the local bandwidth scalars λ_i are defined by:

$$\lambda_i = \left\{ \tilde{f}(X_i)/g \right\}^{-\alpha} \tag{5.5}$$

$\tilde{f}(X_i)$ is a pilot density estimate calculated in (5.1) with h defined in (5.3). g is the geometric mean given by:

$$\log g = n^{-1} \sum_{i=1}^{n} \log \tilde{f}(X_i) \tag{5.6}$$

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

Hardware Implementation

As mentioned earlier, our feature extraction is performed directly in hardware to eliminate the possibility of software tampering. The actual implementation, as shown in Figure 5.9, employs a custom hardware component, i.e., *feature collector*, which is deeply coupled with the CPU and collects the process identifier as well as the features used for process behavior modeling, based on the microprocessor state.

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

Statistics of data dependencies can, then, be generated for instructions in different modes. To this end, we make use of the built-in decoder in the microprocessor to derive the read/write operations on the 4 general purpose registers. A temporary register is associated with each of the general purpose registers, maintaining the `READ` or `WRITE` tag of its last access. For every new access on a general purpose register, its current operation is compared with its last operation, thereby identifying dependency pairs. Accordingly, a counter corresponding to

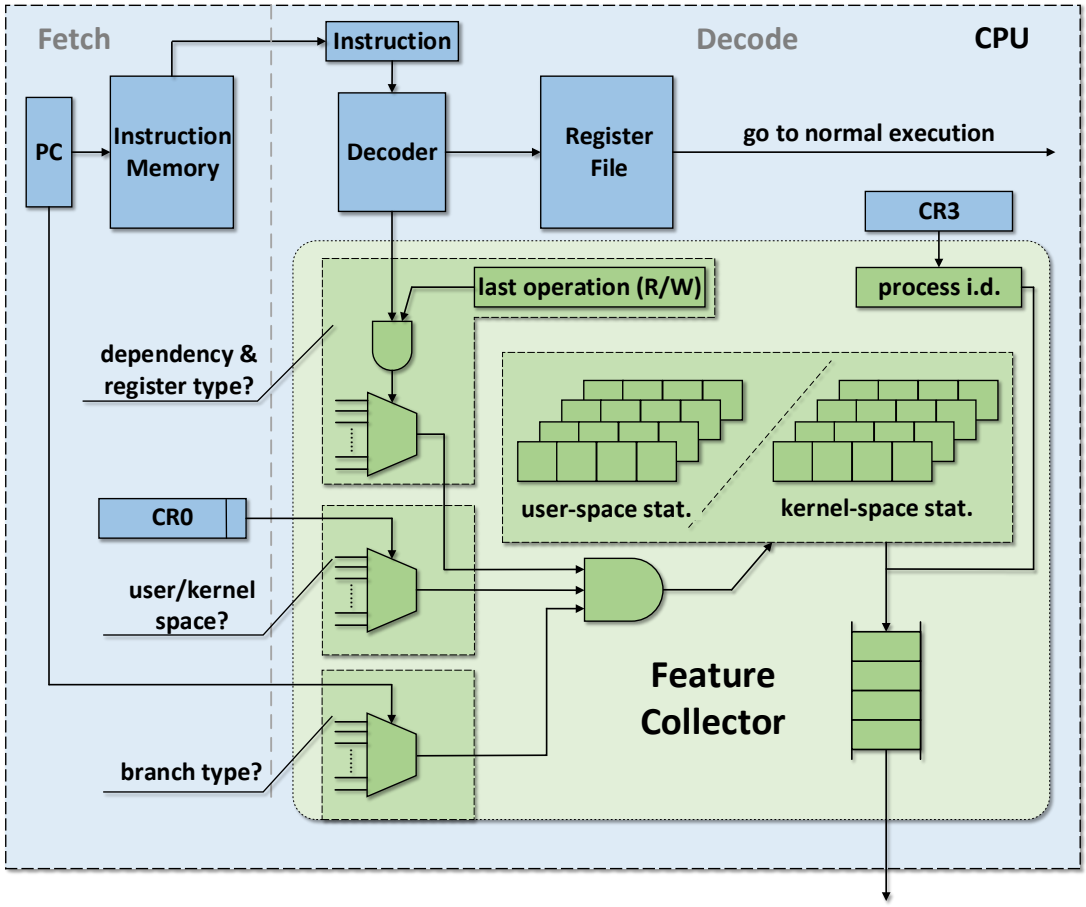


Figure 5.9: Hardware implementation of feature extraction

the specific dependency type is incremented by '1' while the temporary register is updated. Collecting the 3 types of branch statistics, meanwhile, is more straightforward. The feature collector continuously monitors the program counter and instruction operators, in order to detect the occurrence of the branch events, and updates the corresponding counters.

5.3.4 Experimental Results

In this Section, the efficacy of the proposed method in accurately classifying processes and successfully detecting rootkits is evaluated. Additionally, the area/power overhead and logging bandwidth required by the hardware implementation of this method is estimated. Sim-

Table 5.5: Summary of rootkit samples

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

ilarly, the experiments were performed in Simics with same configuration as in the static rootkit detection, while Mibench was used as benign workload. As rootkit samples, the real-world Linux rootkits summarized in Table 5.5 were experimented, which hijack arbitrary kernel service routines to perform denial-of-service attack, file/process hiding, key logging, etc. Implementations of rootkit samples have been elaborated to create more variants, since only a generalized template is provided in the original version.

Effectiveness in Process Identification

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

The process identification results using KNN, SVM and ANN are shown in Table 5.6. The numbers on the left of the slash represent identification accuracy using the testing set excluding rootkit-infected samples, while the numbers on the right side represent the case including these samples. As may be observed, there is no significant difference in the results between the two cases, indicating that rootkit-infected processes did not incur higher misclassification, *even though the training set contained only benign processes.* This observation supports

Table 5.6: Process identification accuracy

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

the conjecture that the rootkit-infected process behavior may not be distinguishable from its benign instances through inter-process behavior deviation. Furthermore, all three classifiers performed well in identifying processes (including rootkit-infected processes), reaching an average accuracy of 99.85%, 99.87%, and 99.87% respectively. This provides solid ground for the next-level rootkit detection.

Effectiveness in Rootkit Detection

Effectiveness of the proposed method in rootkit detection was evaluated separately for each process class. Benign samples in each class were split in half for training and testing, while rootkit-infected samples were only used for testing. The parameters of the KDE algorithm

Table 5.7: Per-process rootkit detection results

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

were optimized independently for each class through cross-validation to maximize rootkit detection capability with minimal false alarms. During the optimization of KDE parameters, only a subset of the rootkit family under test was used, in order to avoid overfitting to the current rootkit dataset, as well as to ensure resilience of the detection model to zero-day rootkit samples.

Table 5.7 summarizes the per-class false positive (FP) (i.e., benign process identified as rootkit-infected) and false negative (FN) (i.e., rootkit-infected process identified as benign) rates. As may be observed, the worst FP rate is 1.8% and the average is 0.75%, while 0% FN rates are achieved for all process classes under test. Indeed, intra-process behavioral models describe process activities at a finer granularity, making rootkit-infected behavior distinguishable. We emphasize that our method outperforms the state-of-the-art hardware-assisted malware detection method, which achieves no significant result in rootkit detection [19]. Furthermore, compared with the software-based counterpart which achieves similarly promising results (i.e., 100% detection rate with low FP rate) [43], our method is inherently more secure since it extracts data in hardware through a custom component. Moreover, it incurs zero runtime overhead due to the non-intrusive collection-to-analysis path. In contrast, the software-based solution incurs a runtime overhead of approximately 3% [43].

Table 5.8: Design overhead of the proposed method

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

Overhead

To evaluate the design overhead of the proposed method, we focus on (i) additional area and power overhead introduced by the feature collector, and (ii) the required data logging bandwidth. Herein, we evaluate the area/power overhead by synthesizing the design of the feature collector using a predictive 45nm Process Design Kit (PDK) [41], which results in area overhead of 649.98 μm^2 and power overhead of 1.9152 mW (at 2GHz). Compared to a 45nm Intel processor[6], the additional overhead incurred by the feature collector is negligible. Furthermore, we ran our workload multiple times to obtain an average estimation of the data logging rate, resulting in a rate of 50.51KB/s. As a point of reference, the performance counter-based method in [19] requires bandwidth of a few hundred KB/s to perform similar analysis. Table 5.8 summarizes the design overhead of the proposed method.

Discussion

Modern microprocessors exploit techniques such as *register renaming* or *Reorder buffer (ROB)* to improve performance. Register renaming *renames* a register to an idle one when a writing request occurs, so that this operation can be executed before its preceding instruction which reads the same register. Similarly, a ROB leverages a register buffer as temporary storage to hold values of speculatively executed instructions. These techniques eliminate anti-dependencies as well as output dependencies and enable out-of-order and speculative

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

program execution. However, they do not affect negatively the proposed method effectiveness, as our feature collector investigates data dependencies when instructions are fetched and decoded in-order, before these techniques are applied.

On the other hand, hardware-based malware detection can be implemented on-chip [37, 27]. This method, however, implements only the feature extraction mechanism in hardware and exports the logged data to a trusted software environment to perform off-chip analysis. In fact, there is a trade-off between on-chip and off-chip solutions. The former generally benefit from prompt reaction to malicious events as compared with the latter; implementing the analysis module on chip, however, increases the design complexity and overhead. Furthermore, when the underlying OS or applications release an update, the configuration of the on-chip analysis module must be updated accordingly, which is not at all straightforward. In contrast, an off-chip analysis module is slower in responding but more flexible, as it can be updated while the on-chip logging component remains unchanged.

5.3.5 Conclusion

In this work, a hardware-assisted infrastructure for performing on-line rootkit detection dynamically was proposed. Compared with the prior work, a new hierarchical detection mechanism is proposed in this method, leveraging intra-process behavior deviation and outlier detection. An incarnation of this idea, which models per-process behavior using data-dependencies, branch statistics and privilege transition, based on which rootkit detection can be performed, was described herein. Experimental results using the Mibench suite with real-world kernel rootkits revealed that almost perfect detection accuracy with very low false positive rate can be achieved. The required logging bandwidth of our method is 50.51KB/s while its hardware cost is negligible compared to the size of a modern microprocessor.

CHAPTER 6

CONCLUSION AND FUTURE WORK

In this dissertation, a line of research has been presented, which proposes hardware-based methodology for performing computer forensics and malware detection. Unlike traditional software-based methods, a hardware-based approach benefits itself from its innate immunity to software tampering, which ensures the security and reliability of the logging and analysis system. A generic architecture is introduced which the hardware-based forensics/malware detection systems need to follow, whose possible implementations can be constructed through combinations of three dimensions (i.e., on-line/off-line, data-centric/program-centric, signature-based/behavior-based), depending on their various purposes. This general concept was illustrated through five incarnations, the first two of which performs a hardware-based workload reconstruction through spatial features and temporal features extracted via TLB profiling while the third of which enables on-line real-time capability in hardware-based workload forensics. Hardware-based static and dynamic detection through system call fingerprinting and intra-process behavior modeling were then evaluated to illustrate the application scenario in malware detection. Experimental results corroborate that low-cost hardware implementations can facilitate highly successful forensics analysis and malware detection.

Regarding the future work, several potential improvements of the current work can be explored, including:

- Mine additional features and methodologies for low-cost real-time workload forensics in resource-restricted scenario, e.g., IoT applications.
- Explore feasibility of finer-grained analysis than the workload identification, e.g., semantic identification.
- Build the entire System-on-Chip (SoC) to facilitate the proposed methodology.

REFERENCES

[1] E51 risc-v core ip. https://www.sifive.com/products/risc-v-core-ip/e51/.

[2] AccessData (2013). Forensic toolkit (ftk).

[3] Almeida, P., C. Baquero, N. Preguica, and D. Hutchison (2007). Scalable Bloom filters. *Information Processing Letters 101*(6), 255 – 261.

[4] ARM (2011). Coresight components techincal reference manual. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0035b/index.html.

[5] ArsenalRecon (2013). Registry recon.

[6] Bengio, Y., P. Simard, and P. Frasconi (1994). Learning long-term dependencies with gradient descent is difficult. In *IEEE trans. on Neural Network*, Volume 5, pp. 157–166.

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

[8] Bloom, B. H. (1970). Space/time trade-offs in hash coding with allowable errors. *Commun. ACM 13*(7), 422–426.

[9] Cabrera, J., L. Lewis, and R. Mehara (2001). Detection and classification of intrusion and faults using sequences of system calls. *ACM SIGMOD Record 30*(4), 25–34.

[10] Carlini, N., A. Barresi, M. Payer, D. Wagner, and T. Gross (2015). Control-flow bending: On the effectiveness of control-flow integrity. In *Proc. of the 24th USENIX Conf. on Security Symp.*, pp. 161–176.

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

[12] Chhabra, R. and V. Nath (2012). Comparative study of Bloom filter architectures. *Global Journal of Researches in Engineering Electrical and Electronics Engineering 12*(4).

[13] Chollet, F. et al. (2015). Keras. https://keras.io.

[14] Christoulakis, N., G. Christou, E. Athanasopoulos, and S. Ioannidis (2016). Hcfi: Hardware-enforced control-flow integrity. In *Proc. of the Sixth ACM Conf. on Data and Application Security and Privacy*, pp. 38–49.

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

[16] Das, S., Y. Liu, W. Zhang, and M. Chandramohan (2016a). Semantics-based online malware detection: Towards efficient real-time protection against malware. *IEEE Trans. on Information Forensics and Security 11*(2), 289–302.

[17] Das, S., Y. Liu, W. Zhang, and M. Chandramohan (2016b). Semantics-based online malware detection: Towards efficient real-time protection against malware. *IEEE Transactions on Information Forensics and Security 11*, 289–302.

[18] Davi, L., M. Hanreich, D. Paul, A. Sadeghi, P. Koeberl, D. Sullivan, O. Arias, and Y. Jin (2015). HAFIX: Hardware-assisted flow integrity extension. In *Proc. of the 52nd Annual Design Automation Conf.*, pp. 1–6.

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

[20] Faiedh, H., Z. Gafsi, and K. Besbes (2001). Digital hardware implementation of sigmoid function and its derivative for artificial neural networks. In *ICM 2001 Proc. The 13th Intl. Conf. on Microelectronics*, pp. 189–192.

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

[22] Garber, L. (2011, January). Encase: A case study in computer-forensic technology. *IEEE Computer Magazine*.

[23] Gers, F. A., J. Schmidhuber, and F. Cummins (2000). Learning to forget: Continual prediction with LSTM. In *Journal Neural Computation*.

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

[25] Hochreiter, S. and J. Schmidhuber (1997). Long short-term memory. In *Nueral Computation*, Volume 9, pp. 1735–1780.

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

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

[28] Kirsch, A. and M. Mitzenmacher (2006). Less hashing, same performance: Building a better Bloom filter. In *Algorithms - ESA 2006*, Volume 4168, pp. 456–467.

[29] Kleen, A. and B. Strong (2015). Intel processor trace on linux. `https://www.halobates.de/pt-tracing-summit15.pdf`.

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

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

[32] Labs, M. (2017). Threats report. `https://www.mcafee.com/au/resources/reports/rp-quarterly-threats-jun-2017.pdf`.

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

[34] Litty, L., H. Lagar-Cavilla, and D. Lie (2008). Hypervisor support for identifying covertly executing binaries. In *17th USENIX Security Symp.*, pp. 243–258.

[35] Maggi, F., M. Matteucci, and S. Zanero (2010). Detecting intrusions through system call sequence and argument analysis. *IEEE Trans. on Dependable and Secure Computing 7*(4), 381–395.

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

[37] Ozsoy, M., C. Donovick, I. Gorelik, N. Abu-Ghazaleh, and D. Ponomarev (2015). Malware-aware processors: A framework for efficient online malware detection. In *IEEE 21st Intl. Symp. on High Performance Computer Architecture*, pp. 651–661.

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

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

[40] Pfoh, J., C. Schneider, and C. Eckert (2011). Nitro: Hardware-based system call tracing for virtual machines. In *6th Intl. Conf. on Advances in Information and Computer Security*, pp. 96–112.

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

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

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

[44] Waterman, A., Y. Lee, D. A. Patterson, and K. Asanovi (2014). The RISC-V instruction set manual. volume 1: User-level ISA, version 2.0. Technical report, California Univ. Berkeley Dept. of Electrical Engineering and Computer Sciences.

[45] Xilinx (2014). LogiCORE IP floating-point operator v7.0 product guide. `https://www.xilinx.com/support/documentation/ip_documentation/floating_point/v7_0/pg060-floating-point.pdf`.

[46] Yeung, D.-Y. and Y. Ding (2003). Host-based intrusion detection using dynamic and static behavioral models. *Pattern Recognition 36*, 229–243.

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

[48] Zhou, L. and Y. Makris (2017). Hardware-based on-line intrusion detection via system call routine fingerprinting. In *Design, Automation and Test in Euro. (DATE)*, pp. 1546–1551.

BIOGRAPHICAL SKETCH

Liwei Zhou was born in Shanghai, China. He joined Tongji University in Shanghai, China in 2007, after completing his schoolwork at Shanghai Experimental School. He received a Bachelor of Science with a major in electrical engineering from Tongji University in June 2011, and entered the electrical engineering graduate program at The University of Texas at Dallas in the same year. After completing his Master of Science in 2013, he continued as a PhD candidate at The University of Texas at Dallas in the same year and started his research in the area of hardware-based system security. In 2018, he received his PhD degree.

CURRICULUM VITAE

Liwei Zhou

Contact Information:

Department of Electrical and Computer Engineering
The University of Texas at Dallas
800 W. Campbell Rd.
Richardson, TX 75080-3021, U.S.A.

Email: lxz100320@utdallas.edu

Educational History:

B.S., Electrical Engineering, Tongji University, 2011
M.S., Electrical Engineering, The University of Texas at Dallas, 2013
Ph.D., Electrical Engineering, The University of Texas at Dallas, 2018

Hardware-based Workload Forensics and Malware Detection in Modern Microprocessors
Ph.D. Dissertation
Department of Electrical and Computer Engineering, The University of Texas at Dallas
Advisors: Dr. Yiorgos Makris

Research Interest:

Applications of machine learning algorithms in hardware-based system security and hardware security

Awards and Honors:

3rd Place in HAC@DAC competition, 2018
3rd Place in TTTC's E. J. McCluskey Best Doctoral Thesis 2018 Award Contest, 2018
1st Place in Cyber Security Awareness Week-Embedded Security Challenge (CSAW-ESC), 2015

Publications:

[1] Liwei Zhou and Yiorgos Makris, "Hardware-assisted rootkit detection via on-line statistical fingerprinting of process execution," in Design, Automation and Test in Europe Conference and Exhibition (DATE), March 2018, pp. 1580-1585.
[2] Liwei Zhou and Yiorgos Makris, "Hardware-based on-line intrusion detection via system call routine fingerprinting," in Design, Automation and Test in Europe Conference and Exhibition (DATE), March 2017, pp. 1546-1551.
[3] Liwei Zhou and Yiorgos Makris, "Hardware-based workload forensics and malware detection in microprocessors," in 17th International Workshop on Microprocessor and SOC Test and Verification (MTV), December 2016, pp. 45-50.

[4] Mohammad-Mahdi Bidmeshki, Gaurav Rajavendra Reddy, Liwei Zhou, Jeyavijayan Rajendran, and Yiorgos Makris, "Hardware-based attacks to compromise the cryptographic security of an election system,"" in IEEE 34th International Conference on Computer Design (ICCD), October 2016, pp. 153-156.
[5] Liwei Zhou and Yiorgos Makris, "Hardware-based workload forensics: Process reconstruction via TLB monitoring," in IEEE International Symposium on Hardware Oriented Security and Trust (HOST), May 2016, pp. 167-172.