A Hardware-Based Architecture-Neutral Framework for Real-Time IoT Workload Forensics

Liwei Zhou, *Member, IEEE*, Yang Hu, *Member, IEEE*, and Yiorgos Makris, *Senior Member, IEEE*

Abstract—Beneath the potential benefits of the rapidly growing Internet of Things (IoT) technology lurk security risks. In this article, we propose a hardware-based generic framework for IoT workload forensics, an infrastructural technique to securely monitor and ensure delivered IoT services in accordance with specifications and regulatory compliance. In particular, this technique identifies digital workloads being executed in real time through dynamic program behavior modeling based on architecture-level data, fulfilled by dedicated machine learning hardware, without the intervention of high-level software, e.g., the OS and/or the hypervisor. In contrast to the conventional software-based solutions, whose effectiveness may be undermined by software attacks, and which introduce significant runtime overhead, a hardware-based framework enables a secure, prompt and non-intrusive solution. The proposed framework was evaluated on Zedboard, a Zynq-7000 FPGA embedding an ARM Cortex-A9 core. Experimental results using Mibench workload benchmark reveal an average workload identification accuracy of 96.37 percent with insignificant area/power overhead.

Index Terms—Hardware-based, forensics, machine learning, security

1 Introduction

The emergence of the Internet of Things (IoT) technology facilitates the integration of multiple physical devices including computers, mobile phones, vehicles, home and industrial appliances, etc., leading to new infrastructures and applications in the domains of transportation, health care, industrial plants, and futuristic robotics, etc. [1]. However, beneath the potential benefits of the IoT technology lurk unprecedented public or private security risks. For instance, a malfunctioning autonomous vehicle would result in massive accidents. Hacked public facilities may incur privacy leakage. The logistics industries could suffer financial loss if the control system malfunctions while, the robotic assistance is unreliable if its embedded system executes unexpected behavior. Thereby, developing digital forensics mechanism to monitor, investigate, and ensure the legitimate execution of the system behavior becomes invaluable.

On the other hand, addressing these security issues for IoT applications, e.g., autonomous vehicle, is not at all straightforward. For example, an evil attacker may degrade the performance of the traffic control system or benefit himself via compromising roadside fog nodes or the targeted vehicle, leading to severe consequences instantly when the compromised vehicle hits the road or the hacked system is deployed. Correspondingly, these security risks imply a *real-time* investigation solution, which is able to monitor the security status of a system continuously to ensure awareness of suspicious behaviors in a timely manner [2]. To this end, we propose real-time workload forensics for IoT applications in order to identify what workloads are executed actively within a system, whose results can facilitate further analyses and reactions.

Intuitively, workload forensics solutions can employ software implementations due to the straightforwardness and flexibility, which can be implemented at OS-level or at hypervisor-level in a virtualized environment. OS-level methods model and analyze the workload behavior by inspecting OS-level semantics, e.g., system-level data structure, file system objects, system call pattern, etc. [3], [4], [5], [6]. Despite their convenience in implementation, the assumption that the underlying OS is trustworthy remains doubtful, since malware may be injected in the OS, running at the same privilege even as the OS, which can subvert the probing and/or analysis software. For example, kernel rootkits may hide or disrupt certain memory pages from being read by software tools or launch denial-of-service attack when detecting the presence of these tools [7]. Moreover, OS-level analysis requires deep coupling with the program execution flow and the OS service control flow in order to obtain and process the target data, thus, introduces notable runtime overhead (commonly 2x to over 10x slowdown), which prevents it from on-line deployment.

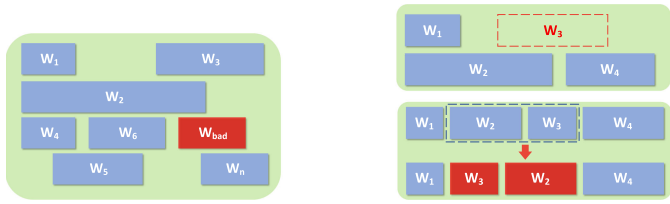
A potential solution to this issue is to leverage Virtual Machine (VM) introspection, i.e., the target OS is wrapped in a VM while the analysis software resides in a

- *Liwei Zhou is with the Electrical and Computer Engineering, University of Texas at Dallas, Richardson, TX 75080. E-mail: zlw_frank110@hotmail.com.*
- *Yang Hu is with the Electrical and Computer Engineering, Erik Jonsson School of Engineering and Computer Science, University of Texas at Dallas, Richardson, TX 75080. E-mail: yang.hu4@utdallas.edu.*
- *Yiorgos Makris is with the Electrical and Computer Engineering, UT Dallas, Richardson, TX 75080. E-mail: yiorgos.makris@utdallas.edu.*

Manuscript received 18 Sept. 2019; revised 5 Feb. 2020; accepted 25 May 2020. Date of publication 5 June 2020; date of current version 8 Oct. 2020. (Corresponding author: Liwei Zhou.)
Digital Object Identifier no. 10.1109/TC.2020.3000237

TABLE 1
Hardware-Based versus Software-Based Forensics

	SW-based		HW-based
	OS-level	Hypervisor-level	
Semantic gap	×	✓	✓
Security	×	×	✓
Runtime overhead	2x - 10x	18% - 9x	~ 0
Platform independence	×	×	✓



(a) Injected new suspicious workload

(b) Bypassed/reordered legitimate workload

Fig. 1. Two application scenarios considered in IoT workload forensics.

higher-privileged hypervisor [8], [9], [10]. As a result, the privileged analysis software can be resistant to OS-level attacks. Nevertheless, the hypervisor itself, as shown through recent work [11], holds several vulnerabilities and can be compromised by intrusion methods. Therefore, analysis tools running on the hypervisor remain exposed to software tampering. On the other hand, although hypervisor-level methods may surpass the OS-level methods in performance due to coupling with lower-level code, their nature as software implementation still lead to significant runtime overhead (18 percent to 9x slowdown).

To address the aforementioned limitations, we propose a hardware-based real-time workload forensics framework. 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. As summarized in Table 1, the proposed framework tries to improve the software-based solutions as follows:

- Employing dedicated hardware tracing mechanism leverages the fact that no software can hide its execution from the hardware, which, therefore, prevents the trace module from software tampering. As a result, a hardware-based solution ensures the integrity and the trustworthiness of the traced data, providing a solid ground for further analysis.
- The hardware-based workload forensics solution, since its deep coupling with the microprocessor, can be deployed on-line with minimal runtime overhead. While software-based solutions incur significant runtime cost and, thus, may be impractical in deployment, a hardware-based solution enables no-intrusive investigation and prompt response to software execution.
- The methodology applied in the proposed framework relies exclusively on common OS and processor architecture characteristics, resulting in an OS-agnostic and architecture-agnostic solution.

The rest of the paper is structured as follows. Section 2 briefly discusses the application scenarios and introduces the corresponding system design, including the critical components, of the proposed framework. The detailed implementation and methodologies applied in each component are introduced in Sections 3, 4, and 5. Section 6 presents the hardware implementation. We evaluate the effectiveness as well as the design overhead of our framework in Sections 7 and 8. Potential limitations are discussed in Section 9. In Section 10, we present the related work. Conclusions are drawn in Section 11.

2 System Overview

2.1 Application Scenario

In this section, we briefly illustrate the application scenarios employing workload forensics. A target IoT system allows execution of a set of legitimate workloads $\{W_1, W_2, \ldots, W_n\}$, which are naturally restricted by the type of devices and corresponding specifications in the IoT network. An adversary is assumed to have access to the physical devices or the network so that he is able to introduce additional functionality or bypass legitimate functionality to benefit himself. Correspondingly, as illustrated in Fig. 1, two common application scenarios are considered herein.

In the first scenario, unexpected suspicious workload W_{bad} can be introduced by additional user-defined programs to benefit his own interest. These programs are not necessary to be specific malicious programs created by an attacker, but can include a combination of common programs which enable new functionalities and violate the pre-defined specification. For example, in an election system facilitated by a homomorphic encryption, an end user can introduce decryption operation and homomorphic summation in the election terminal (which is not allowed in the specification) to contaminate the number of votes of a specific candidate and manipulate the election result [12]. In this case, the decryption and summation workloads introduced intentionally represent the W_{bad}, while the original legitimate data transmission represent the legitimate workloads set.

In the second scenario, an adversary may compromise the target system without introducing additional workloads but through bypassing or reordering legitimate workloads for malicious purpose. For instance, Miller and Valasek has successfully demonstrated vehicle hacking, which can be exploited to disable the brake system or track the car with its built-in navigation system [13]. In this case, illegitimate workload execution flow is created through disabling or reusing existing functionalities.

2.2 System Design

The hardware-based workload forensics framework is required to be capable of distinguishing suspicious workloads from benign ones as well as identifying active workloads in response to the two application scenarios. 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. On the other hand, the data of interest

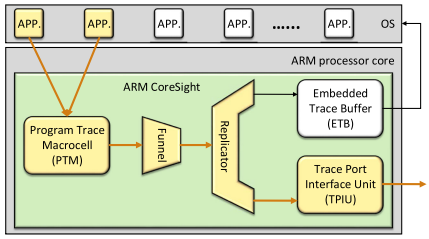


Fig. 2. Architecture of ARM CoreSight.

can be collected in non-intrusive manner, leaving no effect on the original processor execution path. The feature extraction module generates representative features from the collected data, which describe program behavior, while the workload identification module, according to the aforementioned application scenarios, 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.

3 Hardware 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) [14], [15]. Generally speaking, these solutions aim at *non-intrusively* collecting program runtime *branch addresses* so that the dynamic control flow of a program can be 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 Fig. 2.

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 CoreSight 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 only to facilitate the illustration of the proposed concept.

TABLE 2
Summary of Application Scenarios of Commercial Processors With Hardware Tracing Support

Applications	ARM	Intel
Server & 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 m5/m7 family
Embedded applications	Cortex-A35/A17, Cortex-M23/M7/M4	Core i5/i7 family, Xeon E3 family

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 2. 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* [16], 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.

4 Feature Engineering

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 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.

4.1 Spatial Features

We perceive the spatial features as the features which are able to capture the information of the address space layout

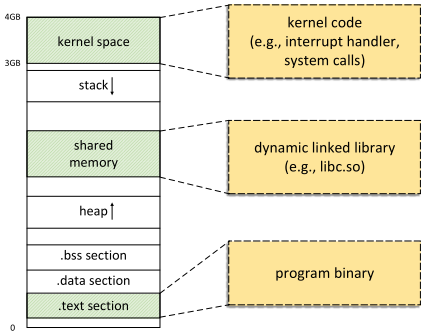


Fig. 3. Typical address space layout.

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 Fig. 3. 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`, the following iterative process runs 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 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.

Listing 1. Heuristic Weighted Partition Algorithm

```
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 ← p_accum + L[u];
if p_accum > mean:
//p contains only one u
if p_accum == L[u]:
E[i] ← u;
total ← total - p_accum;
p_accum ← 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] ← u - 1; //exclude current u
total ← total - (p_accum+L[u]);
p_accum ← L[u];
else:
E[i] ← u;
total ← total - p_accum;
p_accum ← 0;
i ← i + 1;
P_left ← P_left - 1;
if P_left == 1: //the left u will form the last p
break iteration;
return E;
```

4.2 Temporal Features

Essentially, the spatial features introduced in Section 4.1 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.

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 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.

4.3 Real-Time 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, 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.

5 Workload Identification With Anomaly Detection

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 response to the *attack scenario I* (defined in Section 2.1), 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, in response to the *attack scenario II*.

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). Decision tree models a tree-like structure from the feature space and generates a classification rule based on probabilities, where leaf nodes represent a class label and each branch paths from the root to the leaf represents a classification rule. The classification can then be done by searching for the branch with the maximum likelihood. On the other hand, ANN exploits a layered structure, where each layer contains multiple nodes, i.e., *neurons*, which are interconnected with nodes in adjacent layers. Through stacked layers, ANN models an arbitrary function which maps the input layer (feature space) to the output layer (class labels), and thus fulfills the classification. In our scheme, we evaluate DT from the Matlab library and ANN from Keras [18].

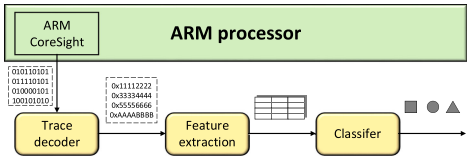


Fig. 4. architectural view of the proposed framework.

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, we employ Recurrent Neural Network (RNN), which has been developed to accept sequential inputs. Essentially, RNN is a variation of the traditional ANN with minor modification in the layered structure. Specifically, in RNN, a self-feedback is applied on each neuron so that its output relies not only on inputs from the last layer but also on the previous computation of its own. By this mean, RNN *memorizes* information of what has been calculated, and therefore, leverages the sequential information in the input sequence. An RNN can be converted into the traditional ANN through unfolding the feedback of its neurons so that the conventional *backpropagation* algorithm can still be applied.

Unfortunately, traditional RNN is known to suffer the *gradient vanishing* problem due to its deep unfolded layered structure, as identified in [19]. Therefore, we employ an alternative architecture of the RNN, namely *Long Short-Term Memory (LSTM)* [20]. LSTM-RNN substitutes the original neuron with a memory cell, whose detailed structure can be found in [21]. By this mean, LSTM maintains a more constant error propagation during backpropagation training so that it enables the RNN to learn over much longer steps, thereby prevents the vanishing gradient. In our implementation, we used LSTM-RNN from Keras [18].

6 Hardware Implementation

In this section, we present the hardware implementation of the proposed framework. Since we leverage the ARM CoreSight to facilitate our design, a custom hardware tracing module is unnecessary. Alternatively, a data decoder is required to decode the trace collected by the ARM CoreSight module. An architectural view of the entire framework, which mainly consists of a trace decoder, a feature extraction module, and a classifier to handle workload identification and anomaly detection, is illustrated in Fig. 4.

6.1 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 [14]. The decoder works at the same frequency as the CoreSight module to synchronize itself with the CoreSight output. Only packets related to the current context ID value and the direct/indirect branch

address, recognized by the predefined specific headers [14], 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.

6.2 Feature Extraction

This component extracts the features from the received branch address sequence for a specific context ID, in parallel with the data stream decoder, once it detects a valid decoded branch address. A feature vector, containing a number of registers (which accords to the number of partitions of integer registers, is instantiated in order to store the CoO. A conditional check is performed on the branch address value in order to access the correct register based on the partition edge derived by the algorithm introduced in Listing 1. A counter that logs the number of processed branch addresses is incremented accordingly and send a signal to the front-end processing unit to stop processing further incoming branch addresses, once the number reaches the pre-defined threshold (which is determined through the evaluation in the following section) or a new context ID is detected.

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 [22], 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. The feature extraction for a current context ID is finalized after the standardization and a `ready` signal is sent to the next-level classifier module.

6.3 Multi-Class 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:

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

where I is the input vector to the ith neuron at jth 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 consists of (1) memory storage that maintains weights and biases of layers and intermediate results, (2) computational logic that fulfills the Equations (1) 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 outputs of the hidden layer which are the inputs of the output layer. 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 [23]. 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.

6.4 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 anomaly 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:

$$class = \begin{cases} suspicious, & if\ max.\ prob. < th(i) \\ argmax, & \text{otherwise.} \end{cases} \tag{2}$$

That is to say, our anomaly detection is implemented by a value comparison between the classifier output and some pre-learned thresholds, in order to filter out potentially illegal program behavior. By this means, the anomaly detection and classification tasks are integrated into one single implementation of the machine learning algorithm, which minimizes the design overhead of the classifier module.

7 Experimental Result

We assess the effectiveness of our method in correctly identifying workloads and filtering suspicious workloads in this

TABLE 3
Summary of Workload Dataset

Class	Description	Benchmark
Automotive /Industrial	programs used in embedded control system, including math abilities, bit manipulation, sorting, and shape recognition algorithms	basicmath, bitcnts, qsort, susan
Server	programs used most frequently on the server side	ls, ps, whoami, id, echo, cat, ifconfig
Telecomm.	programs used in wireless communication, including voice encoding/decoding, frequency analysis, and checksum algorithm	crc, fft, rawdaudio, rawcaudio, toast, untoast
Network	programs used in network devices, including shortest path calculations, tree and table lookups	patricia, dijkstra
Security	programs used for data encryption/decryption and hashing	sha, bf
Consumer	programs used in consumer multimedia devices, including image encoding/decoding and audio encoding/decoding	cjpeg, djpeg, lout, search

section. We illustrate classification results 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 the design overhead.

7.1 Data Collection

Our 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 our feature extraction mechanism in software for evaluation. We use both common Linux commands and MiBench [24], a free commercially representative benchmark suite as our workloads, which include common workload categories in IoT applications, e.g., *automative and industrial control*, *network*, *security*, *telecommunications*, etc. We evaluate 25 program families, summarized in Table 3, while each family is executed with different arguments, creating multiple variations for classification in order to boost the resilience of our framework. For example, program *qsort* sorts different sequences in various length, while program *ls* displayed contents of different directories with multiple options. In total, we collect approximately 400 variations for each program family, which were split randomly in half for training and testing.

7.2 Effectiveness

7.2.1 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 even partitioning method and the heuristic weighted partitioning

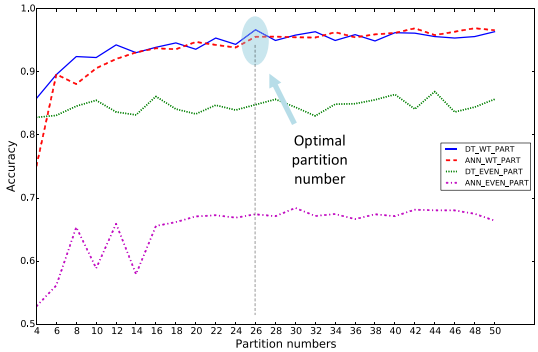
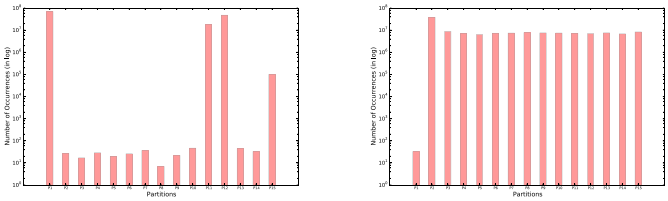


Fig. 5. Average workload identification accuracy according to different partitioning method and partition number.

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. Fig. 5 summarized the average identification accuracy over all the program classes, using both DT and ANN, corresponding to different partitioning solutions. As may be observed, the even partitioning method led to no favorable result (the green and the maple line in Fig. 5) in workload identification, which reached an identification accuracy of approximately 83 percent for DT and 65 percent for ANN. Furthermore, the identification accuracy does not change significantly with the increase in the partition number, which implies that the finer-grain granularity in the even partitioning does not produce a deeper view of program execution.

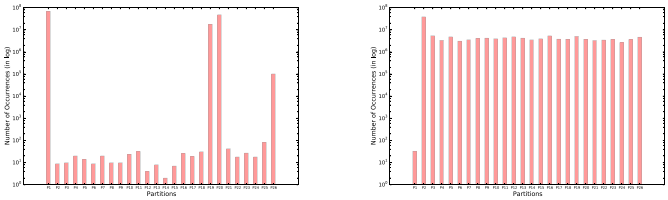
On the other hand, the DT and ANN perform well with the heuristic weighted partitioning method in workload identification. As shown in Fig. 5, the average identification accuracy in both cases (the red and the blue line) increases monotonically according to the partitioning granularity. In particular, the DT obtained an approximately 10 percent gain in the average identification accuracy with finer-grained weighted partitioning, while the ANN obtained an approximately 20 percent gain in the accuracy. Compared with the even partitioning scenario, the DT ultimately surpassed the baseline with approximately 13 percent in performance, while the ANN surpassed the baseline with approximately 30 percent 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 multiple partition number choice for the two partitioning methods was illustrated in Fig. 6, which confirmed the aforementioned implication. Specifically, the distribution corresponding to the even partitioning is illustrated in Figs. 6a, 6c, and 6e (left column), while the distribution corresponding to the weighted partitioning is illustrated in Figs. 6b, 6d, and 6f (right column). As figures in the left column show, the even partitioning leads to a bias distribution, where a majority of the occurrence hits some specific areas. Indeed, as illustrated in Fig. 3, most program instructions are expected to reside within the .text section, shared memory section and the kernel space section. These sections are narrowly small portion of the entire address space, while they are deemed to provide more constructive information. Unfortunately, with even partitioning, most newly-added



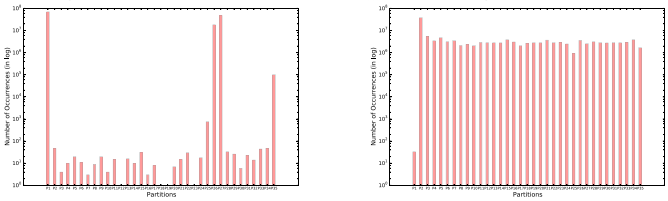
(a) Occurrence (*log* scale) in different partitions when partition P = 15, even partitioning

(b) Occurrence (*log* scale) in different partitions when partition P = 15, weighted partitioning



(c) Occurrence (*log* scale) in different partitions when partition P = 26, even partitioning

(d) Occurrence (*log* scale) in different partitions when partition P = 26, weighted partitioning



(e) Occurrence (*log* scale) in different partitions when partition P = 35, even partitioning

(f) Occurrence (*log* scale) in different partitions when partition P = 35, weighted partitioning

Fig. 6. Distribution of counts of occurrence in the partitions according to even partitioning versus weighted partitioning with different partition size. Weighted partitioning method leads to a uniform distribution. More partitions generate a more informative feature space, hence, results in more accurate classification result.

partitions are assigned to the insignificant area, unrelated to program instructions, since every portion in the address space is treated equally. As a result, more partitions may not reveal additional information within the three critical sections of the address space, hence, contribute less in workload identification.

In contrast, the weighted partitioning takes the aforementioned fact into account. As shown in the right column of the Fig. 6, an uniform distribution is generated, which indicates that the critical areas in an address space are 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 *knee*, 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 *knee* – 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

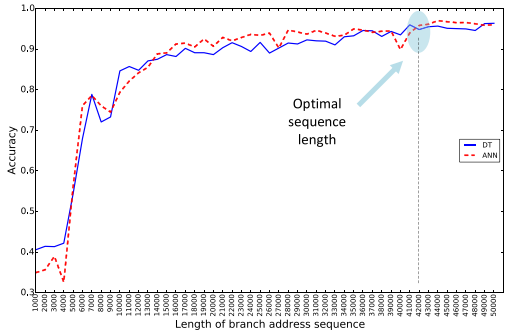


Fig. 7. The average workload identification accuracy according to different sequence length.

average identification accuracy of 96.68 and 95.57 percent for DT and ANN respectively.

7.2.2 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 Fig. 7. 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 *knee* 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 Fig. 8. Specifically, we report the percentage of the optimal length within the average length of the original branch address sequences for each program class. As may be observed, we are able to identify workloads using 49.21 percent of their complete branch address sequence on average, while the best case is 20.9 percent and the worst case is 89.76 percent. Accordingly, with both partition number and sequence length under evaluation optimized, an average identification accuracy of 95.52 percent and 96.37 percent can be reached for DT and ANN respectively.

7.2.3 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 in Sections 7.2.1 and 7.2.2, while the performance of the counterparts using spatial features was considered as the benchmark 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

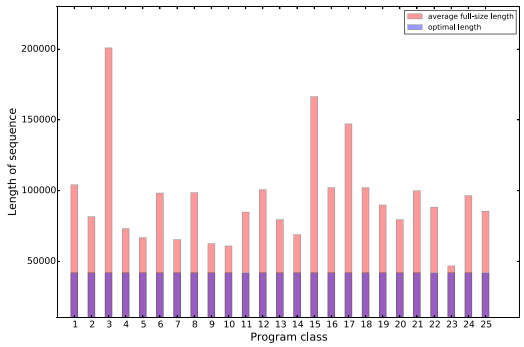


Fig. 8. The optimal sequence length proportional to the full-size sequence length.

95.45 percent for DT and 96.59 percent for ANN. However, the size of the feature space is squared, introducing a dramatic increase in design overhead. We also explored the the 2-gram model with the same size of feature space, i.e., we experimented with a choice of 5 partitions, resulting in 25 features in total. Unfortunately, an average identification accuracy of 90.23 and 91.98 percent 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 percent gain in identification accuracy was obtained. Table 4 summarized the comparison.

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 percent 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 Fig. 7. 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 percent was reached, which achieved approximately 10 percent 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 5 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.

TABLE 4
Effectiveness of 2-Gram Model Versus 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 5
Effectiveness of Partition Sequence Versus 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%

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.

7.2.4 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. Fig. 9 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 well in filtering suspicious programs, reach an average FN rate of 4.5 percent and FP rate of 2 percent 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.

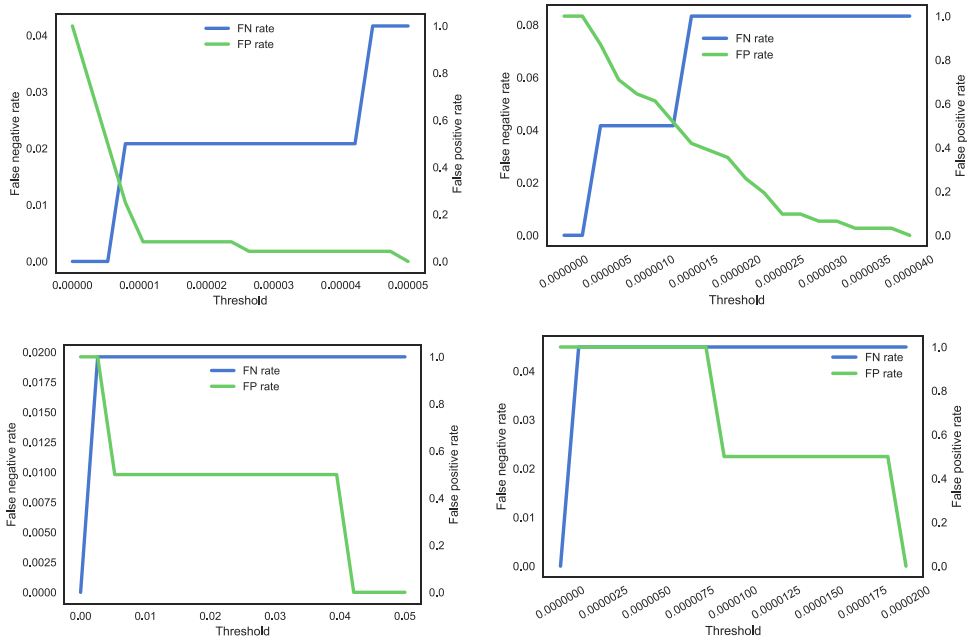


Fig. 9. False negative rate versus false positive rate according to the threshold for different program classes (subset).

8 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.

8.1 Effectiveness

The classification accuracy of the four implementations of the proposed framework is shown in Table 6, 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 decreases 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 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.

TABLE 6
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%

8.2 Overhead Estimation

We evaluate the design overhead of the proposed framework with the implementation derived from Section 8.1 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 7, the entire framework introduced additional use of 1.84 percent LUTs and 0.91 percent 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 percent 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

TABLE 7
Summary of Design Overhead

	Processor & peripheral	Framework
LUT util(%)	12.67%	1.84%
BRAM util(%)	2.14%	1.79%
DSP util(%)	4.09%	0.91%
IO util(%)	32.5%	2.5%
Power (W)	2.072%	0.044%

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 $\div$ BF $\times$ CPI + 234 + 2250 cycles (calculated by the equations defined in Section 6.2 and Section 6.3). Assuming the average BF to be 15 percent (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.

9 Discussion

9.1 Configuration Setup and Update

While a hardware-based solution is advantageous in a prompt response and intrinsic security against the software-based counterpart, it lacks the flexibility in reconfiguration. As illustrated in Figs. 5 and 7, the selection of the sequence length and the partition number in our scheme has great effect on the classification accuracy and potential overhead. However, the optimal values vary case by case, thus, no theoretical optimal value can be studied. The same fact applies to the weights and bias in the neural network design. As a result, these uncertainties leads to difficulties in specifying the underlying hardware design. Nonetheless, our evaluation suggested an empirically good start point, e.g., 42000 of the sequence length that needs a 16-bit register. Such a register, in fact, is able to support a maximum sequence length of up to 65536. Reprogrammability can be introduced further to enable adaptability to different use cases. Practical solutions may include reprogramming the configuration through a secure physical channel or firmware update through secure network, etc. The serial-based neural network design employed in the proposed framework alleviates the difficulty of modifying the neural network structure as well since the number of neurons at each layer can be parameterized and reprogrammed. Nevertheless, the enhancement in the flexibility of hardware-based solutions remains an open question.

9.2 Potential Attack Model

Theoretically, a successful attack to the proposed framework may be launched through contaminating the system functionality or spoofing the detection algorithm. Due to the nature of our hardware-based solution, the former type of attack may assume the access to tamper with the hardware design, or assume the physical access to contaminate the system configuration after the deployment. On the other hand, the latter type of attack requires reverse-engineering of the machine learning algorithm, and proficiency in developing software which is able to spoof the reverse-engineered algorithm while carrying malicious payload. In fact, both attack scenarios have to make relatively strong assumptions about the capability of the adversary, which result in lower probability of real-world attacks.

9.3 ASIC Design

Unfortunately, an IoT system may not always leverage an ARM core embedded with the CoreSight module (or similar hardware tracking module). In such case, as mentioned in Section 3, a custom, CoreSight-equivalent, data tracer is required for data collection. Similar to an In-Target Probe device, the custom tracer needs to be allowed to probe some control registers (e.g., context ID register in ARM architecture or CR3 in Intel architecture) and the program counter of the underlying CPU. A data transmission unit with the probing capability can then become the alternative of the CoreSight module in our scheme and work seamlessly with the other components.

10 Related Work

10.1 Hardware Tracing

The ARM CoreSight hardware tracing module has been involved in various security-oriented research. For example, ARMHEx implements a hardware-based Dynamic Information Flow Trace (DIFT) method based on the ARM CoreSight, which achieves a dramatic reduction in instrument time overhead compared with its software-based counterpart [25]. On the other hand, Ninja develops a malware analysis framework on the ARM processor, which employs the ARM CoreSight to implement the underlying tracing and debugging system [26]. Furthermore, due to its feature of the control flow tracing, the CoreSight module naturally benefits defense solutions to detect *control flow hijacking* attack, e.g., Code Reuse Attack (CRA) [27].

10.2 Hardware-Based System Security

State-of-the-art system security research tends to employ dedicated hardware components due to its innate immunity to software attacks. For instance, a workload forensics method has been proposed in [28], utilizing instructions raising iTLB miss, collected through the hardware, to model the program behavior in order to perform workload forensic analysis. While their work involves knowledge dedicated to x86 instruction set, our proposed methodology, although it leverages ARM CoreSight module in an ARM-based environment, is assumed to be generic since we model the program behavior using architecture-agnostic data.

The possibility of the hardware-assisted malware detection was explored as well, while similar methodologies can be shared. For example, performance counters have been widely adopted to model program behavior, upon which 2-class classification algorithms (rather than multi-class classification) are applied in order to detect malware [29], [30], [31], [32]. Alternative architectural-level information, e.g., instruction opcodes, memory address references, the binary code of system call routines, etc., can also be leveraged to perform the similar analysis [33], [34], [35], [36].

11 Conclusion

We proposed a hardware-based workload forensics framework for IoT system, facilitated by the CoreSight module. Compared with the software-based solutions, our proposed framework maintains immunity to software tampering and enables non-intrusive real-time analysis. 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 parameters used for feature generation (i.e., the partition number and the sequence length) were optimized, leading to an average identification accuracy of 96.37 percent. The hardware implementation was evaluated on the Zedboard FPGA, integrated with an ARM processor, which incurred insignificant design overhead and identification latency.

References

[1] L. Atzori, A. Iera, and G. Morabito,"The Internet of Things: A survey," *Comput. Netw.*, vol. 54, no. 15, pp. 2787–2805, 2010.

[2] C. Huang, R. Lu, and K. R. Choo, "Vehicular fog computing: Architecture, use case, and security and forensic challenges," *IEEE Commun. Mag.*, vol. 55, no. 11, pp. 105–111, Nov. 2017.

[3] B. Dolan-Gavitt, A. Srivastava, P. Traynor, and J. Giffin, "Robust signatures for kernel data structures," in *Proc. 16th ACM Conf. Comput. Commun. Secur.*, 2009, pp. 566–577.

[4] Z. Lin, J. Rhee, X. Zhang, D. Xu, and X. Jiang, "SigGraph: Brute force scanning of kernel data structure instances using graph-based signatures," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2011.

[5] J. Lee, T. Avgerinos, and D. Brumley, "TIE: Principled reverse engineering of types in binary programs," in *Proc. Netwo. Distrib. Syst. Secur. Symp.*, 2011.

[6] A. Slowinska, T. Stancescu, and H. Bos, "Howard: A dynamic excavator for reverse engineering data structures," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2011.

[7] J. Stüttgen and M. Cohen, "Anti-forensic resilient memory acquisition," *Digit. Investig.*, vol. 10, pp. S105–S115, 2013.

[8] L. Martignoni, A. Fattori, R. Paleari, and L. Cavallaro, "Live and trustworthy forensic analysis of commodity production systems," in *Proc. 13th Int. Conf. Recent Advances Intrusion Detection*, 2010, pp. 297–316.

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

[10] Y. Fu and Z. Lin, "Space traveling across VM: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection," in *Proc. IEEE Symp. Security Privacy*, 2012, pp. 586–600.

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

[12] M. Bidmeshki, G. R. Reddy, L. Zhou, J. Rajendran, and Y. Makris, "Hardware-based attacks to compromise the cryptographic security of an election system," in *Proc. 34th IEEE Int. Conf. Comput. Des.*, 2016, pp. 153–156.

[13] C. Miller and C. Valasek, "Remote exploitation of an unaltered passenger vehicle," 2015. [Online]. Available: http://illmatics.com/Remote%20Car%20Hacking.pdf

[14] ARM, "Coresight components techincal reference manual," [Online]. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0035b/index.html

[15] A. Kleen and B. Strong, "Intel processor trace on linux," 2015. [Online]. Available: https://www.halobates.de/pt-tracing-summit15.pdf

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

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

[18] F. Chollet, "Keras," 2015. [Online]. Available: https://github.com/fchollet/keras

[19] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *IEEE Trans. Neural Netw.*, vol. 5, no. 2, pp. 157–166, Mar. 1994.

[20] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Nueral Comput.*, vol. 9, pp. 1735–1780, 1997.

[21] F. A. Gers, J. Schmidhuber, and F. Cummins, "Learning to forget: Continual prediction with LSTM," in *Proc. 9th Int. Conf. Artif. Neural Comput.*, 2000, pp. 850–855.

[22] Xilinx, "Logicore IP floating-point operator v7.0 product guide," 2014. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/floating_point/v7_0/pg060-floating-point.pdf

[23] H. Faiedh, Z. Gafsi, and K. Besbes, "Digital hardware implementation of sigmoid function and its derivative for artificial neural networks," in *Proc. 13th Int. Conf. Microelectronics*, 2001, pp. 189–192.

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

[25] M. A. Wahab, P. Cotret, M. N. Allah, G. Hiet, V. Lapotre, and G. Gogniat, "ARMHEX: A hardware extension for DIFT on ARM-based SoCs," in *Proc. Int. Conf. Field Programmable Logic*, 2017, pp. 1–7.

[26] Z. Ning and F. Zhang, "Ninja: Towards transparent tracing and debugging on ARM," in *Proc. USENIX conf. Secur. Symp.*, 2017, pp. 33–49.

[27] Y. Lee, I. Heo, D. Hwang, K. Kim, and Y. Paek, "Towards a practical solution to detect code reuse attacks on ARM mobile devices," in *Proc. 4th Workshop Hardware Archit. Support Secur. Privacy*, 2015, pp. 3:1–3:8.

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

[29] J. Demme *et al.*, "On the feasibility of online malware detection with performance counters," in *40th Annu. Int. Symp. Comput. Archit.*, 2013, pp. 559–570.

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

[31] M. Kazdagli, V. J. Reddi, and M. Tiwari, "Quantifying and improving the efficiency of hardware-based mobile malware detectors," in *49th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2016, pp. 1–13.

[32] K. N. Khasawneh, N. B. Abu-Ghazaleh, D. V. Ponomarev, and L. Yu, "RHMD: Evasion-resilient hardware malware detectors," in *Proc. 50th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2017, pp. 315–327.

[33] M. Ozsoy, C. Donovick, I. Gorelik, N. Abu-Ghazaleh, and D. Ponomarev, "Malware-aware processors: A framework for efficient online malware detection," in *Proc. IEEE 21st Intl. Symp. HPCA*, 2015, pp. 651–661.

[34] K. N. Khasawneh, M. Ozsoy, C. Donovick, N. B. Abu-Ghazaleh, and D. V. Ponomarev, "Ensemble learning for low-level hardware-supported malware detection," in *Proc. 18th Int. Symp. RAID*, 2015, pp. 3–25.

[35] L. Zhou and Y. Makris, "Hardware-assisted rootkit detection via on-line statistical fingerprinting of process execution," in *Proc. Des. Autom. Test Eur. Conf. Exhibit.*, 2018, pp. 1580–1585.

[36] L. Zhou and Y. Makris, "Hardware-based on-line intrusion detection via system call routine fingerprinting," in *Proc. Des. Automat. Test Eur. Conf. Exhibit.*, 2017, pp. 1546–1551.

Liwei Zhou (Member, IEEE) received the bachelor's degree from Tongji University, in 2007, the MS degree from the University of Texas at Dallas, in 2013, and the PhD degree from the Department of the Electrical and Computer Engineering, University of Texas at Dallas, in 2018. His research interests include trustworthy security-enforced computer architecture for system security applications, e.g., computer forensics and malware detection.

Yang Hu (Member, IEEE) received the bachelor's degree from Tianjin University, China, the MS degree from Tsinghua University, China, and the PhD degree from the Department of Electrical and Computer Engineering, University of Florida, in 2017. He joined the Electrical Engineering Department at UT Dallas, in September 2017. His research interests include cloud and edge computing, NFV, connected vehicles, and heterogeneous architectural support for machine learning. He has published over ten papers at top-tier conferences including ISCA, MICRO, HPCA, ASPLOS, SC and IWQoS, ICCD, PACT, DSN, ICS, etc. His research has been recognized with Best Paper Awards at IEEE CAL 2015 and Best Paper Nominee at HPCA 2017 and HPCA 2018. He serves as program committee and external PC for major computer architecture and system conferences such as HPCA, ISCA, ASPLOS, MICRO, and DAC, ISPASS, ICPP, ICDCS, etc. He also serves as a reviewer for major journals such as ACM Survey, the *IEEE Transactions on Parallel and Distributed Systems*, the *IEEE Transactions on Computers*, the *IEEE/ACM Transactions on Networking*, and IEEE Micro Top picks, etc.

Yiorgos Makris (Senior Member, IEEE) received the diploma degree in computer engineering from the University of Patras, Greece, in 1995, and the MS and PhD degrees in computer engineering from the University of California, San Diego, in 1998 and 2001, respectively. After spending a decade on the faculty of Yale University, he joined UT Dallas where he is currently a professor of Electrical and Computer Engineering, leading the Trusted and RELiable Architectures (TRELA) Research Laboratory, and the Safety, Security and Healthcare thrust leader for Texas Analog Center of Excellence (TxACE). His research focuses on applications of machine learning and statistical analysis in the development of trusted and reliable integrated circuits and systems, with particular emphasis in the analog/RF domain. His serves as an associate editor of the *IEEE Transactions on Computer-Aided Design of Integrated Circuits* and Systems and has served as an associate editor for the *IEEE Information Forensics and Security* and the *IEEE Design & Test of Computers Periodical*, and as a guest editor for the *IEEE Transactions on Computers* and the *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*. He is a recipient of the 2006 Sheffield Distinguished Teaching Award, Best Paper Awards from the 2013 IEEE/ACM Design Automation and Test in Europe (DATE'13) conference and the 2015 IEEE VLSI Test Symposium (VTS'15), as well as Best Hardware Demonstration Awards from the 2016 and the 2018 IEEE Hardware-Oriented Security and Trust Symposia (HOST'16 and HOST'18).