Hardware-Assisted Malware Detection and Localization using Explainable Machine Learning

Zhixin Pan, Jennifer Sheldon and Prabhat Mishra
Department of Computer & Information Science & Engineering
University of Florida, Gainesville, Florida, USA

***Abstract*—Malicious software, popularly known as malware, is widely acknowledged as a serious threat to modern computing systems. Software-based solutions, such as anti-virus software (AVS), are not effective since they rely on matching patterns that can be easily fooled by carefully crafted malware with obfuscation or other deviation capabilities. While recent malware detection methods provide promising results through an effective utilization of hardware features, the detection results cannot be interpreted in a meaningful way. In this paper, we propose a hardware-assisted malware detection framework using explainable machine learning. This paper makes three important contributions. First, we theoretically establish that our proposed method can provide an interpretable explanation of classification results to address the challenge of transparency. Next, we show that the explainable outcome through effective utilization of hardware performance counters and embedded trace buffer can lead to accurate localization of malicious behavior. Finally, we have performed efficiency versus accuracy trade-off analysis using decision tree and recurrent neural networks. Extensive evaluation using a wide variety of real-world malware dataset demonstrates that our framework can produce accurate and human-understandable malware detection results with provable guarantees.**

***Index Terms*—Malware Detection, Explainable Machine Learning, Hardware-assisted Security, Trustworthy Systems**

I. Introduction

Malicious software (malware) is any software designed to harm a computer, server, or computer network and cause damage to the target system. The portability of malware also enables them to proliferate across various platforms at an alarming rate. The recent flood of smart devices and open-source applications provided by unverified third-party developers have created the perfect storm for privacy leakage through malware-infected embedded systems. A recent cybercrime study involving 355 companies across 11 countries covering 16 industrial sectors highlights that malware is the most expensive attack for organizations, with an average revenue loss of $2.6 million per organization in 2018 (11% increase compared to 2017) [1]. Clearly, there is an urgent need to develop efficient malware detection techniques.

Malware detection is a "cat and mouse" game where researchers design novel methods for malware detection, and attackers develop devious ways to circumvent detection. Signature-based detection is one of the most popular commercial malware detection techniques [2]. Signature-based detectors compare the signature of a program executable with previously stored malware signatures. However, signature-based AVS is not useful for unknown zero-delay malware since the respective signature is absent from the database. In fact, signature-based AVS is not effective even for known malware with polymorphic or metamorphic features. These morphic malware have either a mutation engine or rewrite themselves in each iteration through various program obfuscation techniques. While behavior-based AVS is promising in detecting unknown and morphing malware, they are computation intensive. As a result, they are not suitable for resource-constrained systems such as IoT edge devices that operate under real-time, power, and energy constraints.

Recent research efforts explored designing hardware-assisted malware detection with the hardware as a root of trust. The underlying assumption is that, although AVS can be fooled by variations in malware code, it is difficult to subvert a hardware-based detector since the malware functionality will remain the same. There are some promising directions for hardware-assisted malware detection using embedded trace buffers (ETB) and hardware performance counters (HPC). ETB based malware detection [3] shows advantages over HPC based methods [4] in terms of classification accuracy. Despite all these advantages, exploiting hardware components for malware detection is still in its infancy. Machine learning [5] has been successfully used for malware detection [6]–[10]. While hardware-based prediction is promising, it inherits three fundamental limitations:

- These methods predict based on features from individual cycles without considering the malicious behavior that involves interaction between consecutive cycles.
- Since the execution of malware consists of both normal (benign) and malicious computation, they require expensive pre-processing to eliminate useless benign cycles.
- Most importantly, users get only the final decision without understanding how the decision was made or how to localize the malicious activity.

In this paper, we propose a hardware-assisted malware detection that takes advantage of explainable machine learning. It investigates an effective combination of hardware performance counters and embedded trace buffer for efficient detection and localization of malware attacks. It explores the suitability of decision tree and recurrent neural networks for accurate interpretation of classification results. There are success stories in developing explainable machine learning models for computer vision applications. Unfortunately, they are not applicable for malware detection since they consider only static pixel images while we need to handle input data that are time-sequential records. This paper makes the following major contributions:

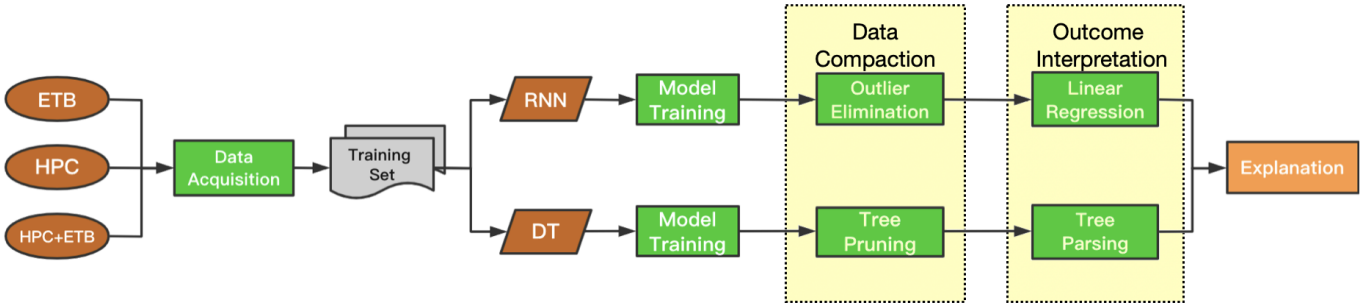


Figure 1: Our proposed malware detection framework consists of four major steps: (i) data acquisition, (ii) model training, (iii) data compaction, and (iv) outcome interpretation.

1) To the best of our knowledge, our approach is the first attempt in developing hardware-assisted malware detection using explainable machine learning, which leads to interpretable detection results as well as improved accuracy compared to the state-of-the-art methods.
2) The interpretation of results sheds light on why the classifier makes incorrect decisions. We show that an effective utilization of hardware performance counters and embedded trace buffer in an explainable framework can lead to accurate localization of malicious behavior.
3) We have evaluated the suitability of decision tree and recurrent neural networks in terms of detection accuracy and hardware overhead. Experimental results using an SoC-based platform running real-world malware benchmarks demonstrate that our approach can lead to accurate detection as well as interpretation of detection results.

The rest of this paper is organized as follows. We survey related efforts in Section II. Section III describes our proposed method on malware detection. Section IV presents experimental results. Finally, Section V concludes the paper.

II. Background and Related Work

The arms race between malware attacks and malware detection has been going on for more than two decades. In the early days, the focus of detection was on static analysis [11], [12]. The basic idea of static analysis is to utilize software filters for malware detection by extracting feature signatures by either machine learning algorithm or human expert knowledge. Unfortunately, this naive approach can be circumvented by obfuscation [13]. Dynamic detection techniques try to defend against obfuscation [6], [14]. Instead of struggling to identify static signatures despite obfuscation, such methods keep track of the runtime behavior of software and analyze/report any malicious behavior such as illegal access. However, both static and dynamic detection methods run on the software level. AVS is unable to detect malware with obfuscation or other deviation capabilities. Moreover, malware can subvert AVS by abusing software vulnerabilities [15].

Researchers have recently turned their interest to hardware-based detection approaches due to their robust resistance against malware attacks compared to software-based detection. Petroni et al. [16] introduced a Peripheral Component Interconnect (PCI) based detector that monitors immutable kernel memory and successfully detects various kernel-level rootkits. Since this PCI-based method relies on physical memory address, it varies from run to run, which makes its performance unstable. Methods using Hardware Performance Counters (HPC) were proposed in [17]–[19], but shortcomings still exist since HPCs involve false positive rates [17] that can still be improved, along with expensive performance penalties incurred by HPC readings. PREEMPT [3] overcomes this weakness by utilizing the Embedded Trace Buffer (ETB), which gives a prediction accuracy as high as 94%. This method was refined in [20] by utilizing deep neural networks, and detection accuracy as high as 98% was obtained. However, none of these approaches are explainable. Therefore, the detection results cannot be interpreted in a meaningful way. Recent studies [21] and [22] have highlighted serious challenges and reliability concerns for HPC-based malware detection. A major objective of our proposed approach is to provide transparency in malware detection [23] that can lead to improved accuracy as well as interpretation of classification results [24], while addressing the HPC reliability concerns as discussed in Section IV-K. To the best of our knowledge, our proposed approach is the first attempt in applying explainable machine learning for hardware-assisted malware detection.

III. Malware Detection using Explainable ML

Our proposed approach enables a synergistic integration of hardware trace analysis and explainable machine learning (ML) for efficient malware detection. Figure 1 shows an overview of our proposed method that consists of four major tasks. The first task performs *data acquisition* from various sources including hardware performance counters (HPC) and embedded trace buffer (ETB). The next task is *model training* that trains a machine learning classifier M using collected traces. We consider both decision tree and recurrent neural network (RNN) as explainable ML models to explore trade-off between accuracy and efficiency. The third task performs *data compaction*. Specifically, we perform outlier elimination and tree pruning for RNN and DT models, respectively. The goal of the last task is to perform *outcome interpretation*. Specifically, the top features ranked by the magnitude of coefficients will provide users the crucial timing information of malware. While simple tree traversal is adequate for DT models, we need to perform linear regression for RNN models. The remainder of this section describes these four tasks in detail.

A. Data Acquisition

We consider diverse hardware features including hardware performance counters (HPC) and embedded trace buffer (ETB). Specifically, we consider data collection from three sources: HPC values, ETB data, and a synergistic combination of them. HPC provides global perspectives since it computes the number of specific events. For example, it can tell us the number of branch mispredictions. Even if we have only two branches (if statements) in the program (inside a loop), we do not know which of them was mispredicted how many times or if any of them was predicted correctly most of the time. In contrast, if we are tracing the branch condition of a specific branch in an ETB, we can identify how many times the branch condition was true and/or how many times a specific branch was taken. Due to hardware overhead constraints, the ETB size is limited. Therefore, we cannot trace conditions of all branches or other signals (events) in an ETB [25], [26]. In reality, we have to rely on both HPC and ETB values for malware detection.

The frequency of data collection provides a trade-off between accuracy and efficiency. Higher frequency of data collection can lead to unacceptable performance degradation. On the other hand, infrequent data collection can lead to delayed detection (damage has been done already) or even possibility of not detecting at all (footprint unavailable).

There are two important considerations in selecting HPC values. (1) Modern processors support numerous performance counters. We need to select the ones that are relevant for malware detection such as number of instructions, number of exceptions, context change, bus accesses, etc. (2) HPC values can lead to high false positive rate since it is easy for a sophisticated malware to alter the global statistics (e.g., executing redundant instructions). Instead of using the actual values, we compute the differences of HPC values at different time intervals. In this work, we recorded various hardware performance counters (HPCs) including branch-instructions, branch-misses, cache-misses, cache-references, cpu-cycles, instructions, stalled-cycles-backend, stalled-cycles-frontend, L1-dcache-load-misses, L1-dcache-loads, L1-dcache-store-misses, L1-dcache-stores, L1-icache-load-misses, branch-loads, branch-load-misses, dTLB-load-misses, dTLB-store-misses, and iTLB-load-misses. We have uploaded these dataset at *https://github.com/Jshel/MalwareJournalExtension* that can be used by researchers to reproduce the experimental results.

Most system-on-chip (SoC) designs have an in-built embedded trace buffer that stores values of selected signals during runtime. Trace buffers are widely used for post-silicon debug [26]. Due to design overhead considerations, the size of the trace buffer is limited. For example, a typical trace buffer can store values of 1024 signals for 4096 clock cycles in a design with multi-million signals. In some systems, the trace signals are selected during design time based on specific observability constraints. Many modern systems allow dynamic signal selection where we can choose a small set of beneficial signals (e.g., 1024 in the above example) from a set of choices. The collected traces can be viewed as a $w \times d$ table, where w is the *width*, d is the *depth* of the trace buffer.

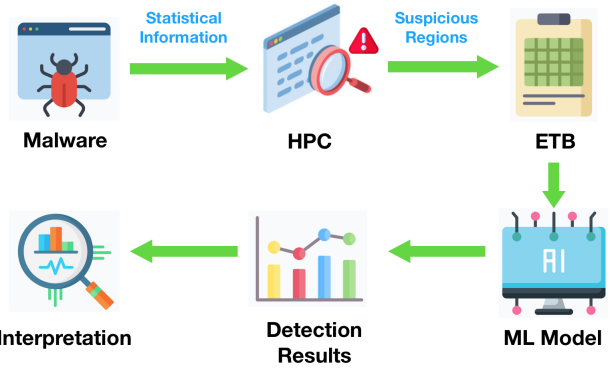


Figure 2: Malware detection using both embedded trace buffer (ETB) data and hardware performance counter (HPC) values.

In other words, it represents the recorded values of w traced signals over d clock cycles. We split each column as a single feature component, i.e values of all traced signals within one single cycle. Next, we apply explainable machine learning for malware detection.

HPC values provide global view of the system that can be utilized to detect malicious behaviors. However, HPC values cannot be trusted in pinpointing the malicious activity. On the other hand, ETB data provides local perspectives in the form of values of specific signals across time. As a result, ETB data is suitable for localization of a malicious activity. Clearly, an effective combination of HPC and ETB would be ideal for efficient malware detection and localization. Figure 2 shows one specific way of utilizing both HPC and ETB. In this scenario, HPC values provide coarse-grained suggestions (timeframe) of malicious activity, which can be confirmed (or denied) by ETB-based fine-grained analysis.

B. Model Training

We consider two types of ML models in our framework, Decision Tree (DT) and Recurrent Neural Network (RNN). We have selected DT because it is a self-explained ML model which has a natural compatibility with explainable ML. The working process of DT is performed by a sequence of binary decisions to reach a conclusion (leaf node). Therefore, when this process completes, the reason for the classification results can be illustrated by the path from root to the leaf. For example, in Figure 10, we can start from a leaf node (e.g., Malware) and trace back to the root to know the sequence of events that led to this decision. We have also selected RNN model due to its outstanding capability in handling time-sequential data as shown in Figure 3. This is based on the observation that the malware detection framework should focus on analyzing information within certain time periods, instead of merely taking overall statistics into account. These two models also provide a trade-off between the prediction accuracy and hardware overhead – DT introduces low overhead but RNN model provides higher accuracy.

1) Training of Decision Trees (DT): The training process of DT is rather straightforward due to its simplicity. First, we collect data and mark it as the total set of samples S, where each sample represents data trace of one individual program

(either benign or malicious). For each sample, the recorded hardware events (from HPC) or register value (from ETB) are considered as attributes stored in a list L. Then we follow the traditional workflow to generate the DT model by recursively selecting suitable attributes until convergence.

A fundamental problem in design of DT model is how to determine the attribute as the bisector at each step. In our task, we use Iterative Dichotomiser 3 (ID3) [27] algorithm to solve this problem. The basic idea is to make every leaf node as 'pure' as possible, i.e, either contains all benign or malicious program samples. If we represent benign samples as '1' and malicious samples as '0', the 'purity' measurement is referred to as *entropy*. Using the principles of information theory, the entropy (Ent) can be computed as follows:

$$Ent(S) = -p_+ log_2 p - p_- log_2 p_-$$

where p_+, p_- represents the proportion of benign/malicious data samples, respectively.

Assumes that S is a set containing 10 examples with 10 benign samples and 0 malicious ones. Then, the entropy of S is computed as 0, and vice versa for all malicious case. Instead, if S is mingled with both types of data, the entropy will be greater than 0. Therefore, entropy intuitively represents the 'chaotic' nature of the current data set. *The primary objective of the model training is to reduce the total entropy.* Therefore, the best bisector should lead to the largest decrease of entropy in the system, which is referred as 'information gain' in information theory.

Based on the above observations, Algorithm 1 outlines the training process of DT. It is a greedy algorithm that grows the tree top-down. At each node, it tries all possible attributes and compute the decrease of entropy. We select the best attribute to bisect the node, and this process continues for its children. Finally, a tree with each leaf node's entropy being 0 demonstrates the convergence of the process.

Algorithm 1 Decision Tree Generation

Input: Data sample set S, attribute list L

procedure DT_GEN(S)
  $l^* \leftarrow NULL$, $e^* \leftarrow 0$
  if $Ent(S) = 0$ **then return** S
  end if
  for each $l \in L$ **do**
    $\{S_1, S_2\} = bi\_sect(S, l)$
    if $|Ent(S_1) + Ent(S_2) - Ent(S)| \geq e^*$ **then**
      $l^* = l$
      $e^* \leftarrow |Ent(S_1) + Ent(S_2) - Ent(S)|$
    end if
  end for
  $\{S_1, S_2\} = bi\_sect(S, l^*)$
  $S.child_1 \leftarrow$ DT_GEN(S_1)
  $S.child_2 \leftarrow$ DT_GEN(S_2)
  return S
end procedure

2) Training of RNN Models: A straightforward way to implement the framework is to directly feed data traces to a fully-connected DNN to make predictions. However, this type of model cannot handle time-sequential data. As discussed previously, an ideal hardware-assisted malware detection technique should be able to make predictions based on data samples collected from multiple cycles. This is due to the fact that it monitors the behavior of software at runtime, where relying on single-cycle data is not effective since malicious behavior usually consumes several sequential cycles. Moreover, single-cycle based strategies are likely to mispredict a benign software as malicious. This is due to the fact that malware also contains normal operations, and considering these benign operations as important features of malware can lead to misclassification. A well-designed pre-processing strategy can mitigate this mistake by filtering overlapped common behaviors shared by both malicious and benign programs. However, it is difficult to design such a strategy. Moreover, there is no guarantee that it can be performed in all cases. In summary, an ideal machine learning model for our task should satisfy the following two properties:

1) Ability to accept time series data as input.
2) Ability to make decisions utilizing potential information concealed in consecutive adjacent inputs.

We propose to utilize *Recurrent Neural Network (RNN)* training to tackle this problem. Algorithm 2 outlines the training procedure. In order to explain the working principles of the algorithm, we need to describe RNN as well as its importance in the context of malware detection. RNN is powerful in handling sequential input data. A classic structure of RNN is shown in Figure 3.

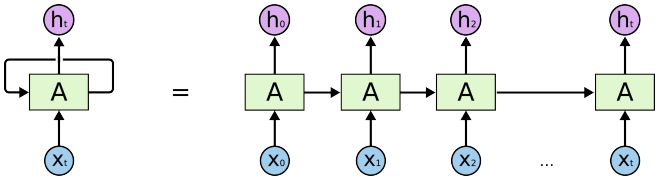


Figure 3: Recurrent Neural Network (RNN)

In the picture, **A** represents the neural network architecture, where $x_0, x_1, x_2, ..., x_t$ represents the time series inputs and h_is are the outputs of hidden layers. As we can see from the left-side of the figure, instead of finishing the input-output mapping in one forward pass, the RNN accepts sequential inputs. For each single input x_i, RNN not only provides immediate response h_i, but also stores the information of the current input by updating the architecture itself. On the right-side of the figure, information corresponding to the previous step will also be fed into the architecture to supply extra information by unrolling the RNN structure. For malware detection using trace data, we can directly set each column of trace table as inputs, and accept the hidden state of final stage, i.e h_t, as the final output.

Due to this recurrent structure of RNN, it suffers from the *vanishing gradient* [28] and *exploding gradient* [29] problem. Specifically, when performing *back-propagation* [30], if the initial gradient is less than 1, then the gradient at the last

moment will disappear and vice versa. Both situation will lead to failure in training process. To tackle this problem, we use a specific type of RNN model, *long-short term memory* (LSTM) [31]. LSTM adopted a *gate* mechanism [32] to solve vanishing gradient and exploding gradient. Meanwhile, the gate mechanism provides feature filtering, saving useful features and discarding useless features, which greatly enriches the information representation capacity of the model.

LSTM is suitable for handling time sequential data, but it is not guaranteed to learn features from adjacent inputs. For time series inputs, considering inputs in groups and training the model to make decision based on co-occurrence of sequential features is crucial. This can be achieved by appending a penalty term to the loss function. This term can force models to group adjacent elements together from input feature map. The loss function with penalty for model (Figure 3) can be written as:

$$J = \frac{1}{N}\sum_{i=1}^{N} L(\mathbf{A}(\mathbf{x}_i), \mathbf{y}_i) + \frac{\lambda}{2}\sum_{k=1}^{t} ||h_i - h_{i-1}||$$

where $\mathbf{A}$ is the model, $\mathbf{x}_i$ is a training sample, the label of $\mathbf{x}_i$ is denoted as $\mathbf{y}_i$, the total number of training samples in a batch is denoted as N, and L is the dissimilarity measurement which is frequently selected to be cross entropy for classifiers. Aside from these regular terms, we introduce a penalty term $\sum_{k=1}^{t} ||h_i - h_{i-1}||$, which tries to minimize the difference between the hidden state outputs of each time step. This is to restrict the impact brought by one single clock cycle input, and prevent the machine learning model from updating its inner feature map too significantly unless it produces a relatively long sequence of similar pattern. Based on the assumption that malicious behavior happens in multiple sequential cycles instead of just one, this training scheme enables LSTM to take adjacent inputs as groups for gathering information and making decisions. The training procedure is outlined in Algorithm 2.

Algorithm 2 Training process of the LSTM Model

```
for each iteration do
    σ = 0
    for i = 1 to t do
        compute adjacent difference Δh = |h_i − h_{i−1}|
        σ+ = Δh
    end for
    Add σ to loss function J
    Compute gradient of modified loss function ∇J
    Update model parameters Θ = Θ + ∇J
end for
```

C. Data Compaction

Once we have the trained ML models, we perform the data compaction to improve detection efficiency. Due to inherent differences in ML models, the compaction procedures are also different between decision tree and RNN (LSTM) models. While compaction can be viewed as *outlier elimination* in RNN models, it is essentially *tree pruning* for decision trees. The remainder of this section describes these steps in detail.

1) Outlier Elimination in RNN Models: Once we have the well-trained model, we can start to perturb the target input $\mathbf{x}$ to generate corresponding perturbed output dataset $\mathbf{Y}$. This is achieved by randomly flipping several bits in target input $\mathbf{x}$. However, the raw output $\mathbf{Y}$ cannot be directly applied to regression algorithm in the next step. This is due to the fact that random perturbation can generate anomalous data, such as data points with extreme values. These data points are isolated from the others in the cluster so they can introduce huge deviation in regression algorithms. To address this, we need to efficiently remove isolated points in $\mathbf{Y}$.

We deployed a random voting algorithm to achieve this goal. The basic idea is to cut a data space with a random hyperplane, and two subspaces can be generated at a time. We continue to randomly select hyperplane to cut subspaces obtained in the previous step, and the process continues until each subspace contains only one data point. Intuitively, we can find that those clusters with high density will not be entirely dismembered until they are cut many times, but those in the low density regions are separated much earlier.

Figure 4 shows an illustrative example. If we want to isolate x_0, we need to draw l_1, i.e cut the space one time, while x_1 needs a lot more partitions. So x_0 is more likely to be an outlier than x_1. Note that the process of cutting space can be naively represented by a binary tree as shown in Figure 4.

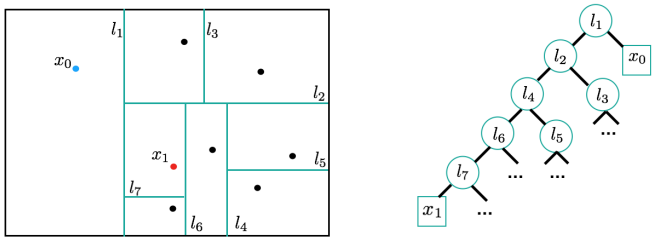


Figure 4: Finding outliers

In general, a threshold θ is applied to categorize isolated and clustered points. For each data point, we check the depth of it inside the binary tree. A point is considered as isolated once the depth exceeds the given threshold. These isolated points are more likely to be extreme value data points and should not be used by the regression algorithm. Eliminating them is likely to improve the accuracy of regression. In order to ensure reliability, we repeat this procedure for several times to obtain a forest of trees, and let them vote for the final decision. The pseudocode for this forest voting algorithm is shown in Algorithm 3.

2) Tree Pruning in Decision Tree: For DT, the outlier elimination work is done by 'pruning', the shortening of branches of the tree. Pruning is the process of reducing the size of the tree by turning some branch nodes into leaf nodes, and removing the leaf nodes under the original branch. Pruning is useful because classification trees may fit the training data well, but may do a poor job of classifying new values. Lower branches may be strongly affected by outliers. Pruning enables us to obtain a streamlined tree and minimize the problem. A simpler tree often avoids over-fitting.

Algorithm 3 Outlier Elimination

Input: Data points set V, threshold θ

procedure DT_BUILD(V)
  $root \leftarrow V$
  $V_1, V_2 \leftarrow$ *bi-sect*(V)
  if $size(V_1) = 1$ **then return** $child_1 \leftarrow V_1$
  else $child_1 \leftarrow$ DT_BUILD(V_1)
  end if
  if $size(V_2) = 1$ **then return** $child_2 \leftarrow V_2$
  else $child_2 \leftarrow$ DT_BUILD(V_2)
  end if
  return $root$
end procedure

procedure FORESTVOTING(θ)
  $res \leftarrow \emptyset$
  for each point v in V **do**
    $cnt \leftarrow 0$
    for each isolated tree t **do**
      $d \leftarrow$*depth of v in t*
      if $d \geq \theta$ **then**
        $cnt \leftarrow cnt - 1$
      else
        $cnt \leftarrow cnt + 1$
      end if
    end for
  end for
  if $cnt \geq 0$ **then**
    $res \leftarrow res \cup \{v\}$
  end if
  return res
end procedure

D. Outcome Interpretation

After obtaining the compacted classification results, the next step is to analyze the reason for classifiers to produce such outputs. For DT models, due to its self-explainability, this task is performed by a trivial tree traversal. However, for LSTM based RNN, there is no such convenient way. Therefore, we apply a linear regression based approach to achieve outcome interpretation. As expected, the tree traversal is fast and simple, but it offers less information compared to LSTM models. While LSTM is more expensive, it can provide detailed information on malware's behavior compared to DT models. The remainder of this section describes these two methods in detail.

1) Tree Traversal in DT Models: A decision tree is a tree-like graph with nodes representing the place where we pick an attribute and ask a question; edges represent the answers to the question; and the leaves represent the actual output or class label. They are used in non-linear decision making with simple linear decision surface. Decision trees classify the examples by sorting them down the tree from the root to some leaf node, with the leaf node providing the classification to the example. Each node in the tree acts as a test case for some attribute, and each edge descending from that node corresponds to one of the possible answers to the test case. This process is recursive in nature and is repeated for every subtree rooted at the new nodes. DT is designed to answer a particular question, and the top-down traversal of the tree explicitly explains how users make decisions. There are many conditions to check, and a user needs to take all these factors into account.

The explanation provided by DT is based on a hidden assumption: the logic behind classifier is defined to be a sequential selection. In real-life applications, this assumption is vulnerable and can lead to misprediction, which will be demonstrated in Section IV.

2) Linear Regression in LSTM-based RNN Models: Compared to DT, there is no built-in property to provide explanation. Therefore we apply a linear regression based approach. A linear regression algorithm allows us to approximate locally nonlinear relationship with proper precision. Formally, given a data set $\{\mathbf{y}, \mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_n\}$, where n is the number of samples, linear regression takes the following form by appending error variable ϵ:

$$\mathbf{y} = \sum_{i=1}^{n} a_i \mathbf{x}_i + \epsilon$$

where a_is are model parameters, and the goal is to minimize ϵ as much as possible. The simplest scenario occurs when $\mathbf{y}$ and every $\mathbf{x}_i$ are real numbers. In our case, the input is the $w * d$ trace table $\mathbf{X}$ as mentioned before. Since we treat each column of this table as an individual input feature, we have $\mathbf{X} = [\mathbf{x}_1\, \mathbf{x}_2 \ldots \mathbf{x}_d]$, where each $\mathbf{x}_i$ is a vector in the size of $w * 1$. We choose $\mathbf{y}$ as the output of last hidden state, i.e h_t in Figure 3 which is also a $w * 1$ vector. This leads to an optimization problem:

$$\arg\min_{\mathbf{a}} ||\mathbf{X}\mathbf{a} - \mathbf{y}||_2$$

where $\mathbf{a} \in \mathbb{R}^d$ is $[a_1\, a_2 \ldots a_d]^T$, i.e, coefficients to be solved. This is a common convex optimization problem and its solution can be obtained by *least squares* which gives

$$\mathbf{a} = (\mathbf{X}^t\mathbf{X})^{-1}\mathbf{X}^t\mathbf{y}$$

Unfortunately, this method cannot be directly applied to solve our task. First, this theoretical solution exists only when $\mathbf{X}^t\mathbf{X}$ is invertible (full rank), which is not satisfied for most of the time. Second, even when $\mathbf{X}^t\mathbf{X}$ is full rank, linear regression assumes input vectors are *independent*, otherwise it will produce unreliable results when any two of $\mathbf{x}_i$ (columns) are highly correlated. Specifically, assume that the regression function is computed to be $\hat{\mathbf{y}} = a\mathbf{x}_1 + b\mathbf{x}_2 + c\mathbf{x}_3 + d$, where $\mathbf{x}_1$ and $\mathbf{x}_2$ are highly related features and they are very close to each other. Then there is a canceling effect between a and b. Increasing a by certain amount while decreasing b by the same amount at the same time will not lead to drastic change in $\hat{\mathbf{y}}$. This can cause high variance of computation results for coefficients. For example, if $x_1 \approx x_2$, then regression functions $\hat{\mathbf{y}} = \mathbf{x}_1 + \mathbf{x}_2 + c\mathbf{x}_3 + d$ and $\hat{\mathbf{y}} = 101\mathbf{x}_1 - 99\mathbf{x}_2 + c\mathbf{x}_3 + d$ can reach the same level of accuracy. The problem becomes ill-posed since absolute value of a and b can vary significantly under different computing procedure or initial conditions. Then the comparisons between $|a|$, $|b|$ and $|c|$ are not useful, therefore, the interpretability of the model is greatly reduced. Since adjacent columns in trace table are sequential records

of signal values within a short duration, violation of this independence assumption is likely to happen.

In our study, we have applied *ridge regression*, which is an improved least squares estimation method, and the fitness of correlated data is stronger than general regression. Ridge regression is achieved by appending one extra penalty term to the optimization problem:

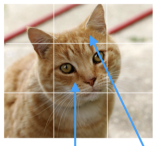
$$\arg\min_{\mathbf{a}} ||\mathbf{Xa} - \mathbf{y}||_2 + \lambda||\mathbf{a}||_2$$

Intuitively, a size constraint is imposed to restrict the absolute value of all coefficients, which alleviate the problem of high variance of coefficients. Moreover, notice

$$\arg\min_{\mathbf{a}} ||\mathbf{Xa} - \mathbf{y}||_2 + \lambda||\mathbf{a}||_2 \rightarrow \arg\min_{\mathbf{a}} ||(\mathbf{X} - \lambda\mathbf{I})\mathbf{a} - \mathbf{y}||_2$$

Replacing $\mathbf{X}$ with $\mathbf{X} - \lambda\mathbf{I}$ is a general way to avoid the problem for $\mathbf{X}$ being singular matrix. Also, data was centralized and the problem of high variance is alleviated. Therefore, with ridge regression, coefficients obtained is more reliable and fit better for our dataset, which has high correlation.

Once coefficients of regression are obtained, we can derive the importance ranking, then interpret it into meaningful information in the context of malware detection. Figure 5 illustrates the way of interpretation, and demonstrates how similar it is to the counterparts in computer vision domain.



$l_1(\mathbf{x}) = a_1x_1 + a_2x_2 + ...$

(a) Blocks of the cat's face and ear distinguish it from other categories.

Buffer/Cycle	1	2	3	4	5	6	7	8
CS	C_1	A_1	C_3	A_3	C_5	A_5	C_7	A_7
AS	A_1	A_1	A_3	A_3	A_5	A_5	A_7	A_7
ES	E_1	B_1	B_1	E_4	B_4	B_4	E_7	B_7
DS	D_1	E_1	B_1	D_4	E_4	B_4	D_7	E_7
BS	B_1	B_1	B_1	B_4	B_4	B_4	B_7	B_7
T1	C_1	A_1	C_3	A_3	C_5	A_5	C_7	A_7
T2	D_1	E_1	B_1	D_4	E_4	B_4	D_7	E_7

$l_2(\mathbf{x}) = b_1x_1 + b_2x_2 + ...$

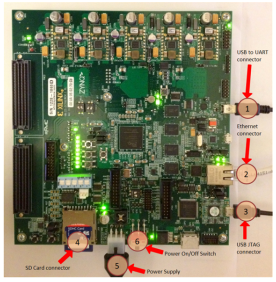
(b) Malicious behavior most likely to happen in the sixth and first cycle.

Figure 5: Comparison between interpreting malware detection results and image classification task in previous example.

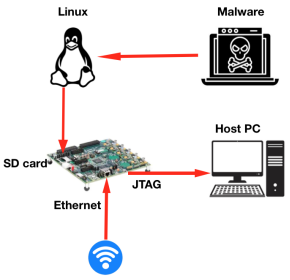
As we can see from the figure, the top features come with large coefficients that are likely to be related to the malicious behavior. Next, we can check the clock cycle distribution of these top features. It is expected to provide us with extra information about the malware. For example, if we observe adjacent cluster of top features, then the time slot within which they reside shall provide an indication of time information about when malicious behavior happened. Similarly, if clock cycle numbers are periodically separated, the detected malware is likely to repeat its malicious activity periodically. Typical malware acting like this usually works in a client-server mode, where client program steals private data and sends message to the hacker's server in a periodic interval. For a closer look, we can split the table into rows and go through the same process. This will lead to identification of trace signal values that are most likely relevant to the malicious behavior, which will lead to malware localization as demonstrated in the next section.

IV. Experiments

This section is organized as follows. First, we outline the experimental platform and describe various malware and benign benchmarks. Next, we discuss various data acquisition techniques. Finally, we present results in terms of malware detection accuracy, time efficiency, and outcome interpretation.



(a) ZYNQ SoC Board



(b) Platform Layout

Figure 6: Experimental platform using SoC board

A. Experimental Setup

We ran malicious and benign programs on the Xilinx Zynq-7000 SoC ZC702 evaluation board as shown in Figure 6. This board integrates double ARM Cortex-A9 cores. We installed $xilinx - zc702 - 2017\_3 : 4.9.0 - xilinx - v2017.3$, a Linux kernel image for the ZC702 evaluation board generated using PetaLinux, to the board using the provided 8 GB SD card. To view the contents of internal signal values, we link the board to the System Debugger in Xilinx SDK version 2017.3, which uses a hardware server to allow us to compile and run these programs on the board while monitoring traced signal values. This configuration involves connecting the board to a host computer running Xilinx SDK using Ethernet (to run programs using the System Debugger) and UART (to interface with the embedded OS to obtain HPC values and set up the Ethernet connection allowing the SDK to gather ETB values).

B. Malware and Benign Benchmarks

In this study, we consider a wide variety of malware families [33] including the following three popular ones: *Bashlite Botnet*, *PNScan Trojan* and *Mirai Botnet*. Bashlite, also known as Gafgyt or LizardStresster, is a malware family targeting Linux systems. Bashlite infiltrate in IoT devices and these poisoned devices will be used to manipulate large-scale distributed denial-of-service (DDoS) attacks. It uses Shellshock to gain a foothold on vulnerable devices, then remotely executes commands to launch DDOS attacks and download other files to the compromised device. It works in a client-server mode where poisoned devices keep sending requests to a remote server checking for possible update releases or malicious requests. PNScan is an open source Linux Trojan which can infect devices with ARM, MIPS, and PowerPC architectures. This Trojan or applications with this Trojan embedded can invade network devices. This malicious program has only one goal – obtain the router's access password through brute force. If the intrusion is successful, the Trojan will load a malicious script into the router which will download the corresponding backdoor program according to the router's system architecture. Mirai is an upgraded variant of Bashlite. Mirai can efficiently scan IoT devices and infect fragile devices like the ones encrypted with default factory settings or weak

Table I: Comparison of detection accuracy with different acquisition approaches for Decision Tree (DT)

	HPC			ETB			HPC+ETB					
Benchmarks	**Accuracy** (%)	**FP** (%)	**FN** (%)	**Accuracy** (%)	**FP** (%)	**FN** (%)	**Accuracy** (%)	**FP** (%)	**FN** (%)	**F1 Score**	**improv./HPC** (%)	**improv./ETB** (%)
Bashlite	80.3	9.5	10.2	84.7	7.3	8.0	88.0	5.9	6.1	0.88	7.7	3.3
Mirai	69.8	16.4	13.8	71.2	10.4	18.4	84.1	6.6	9.3	0.84	14.3	12.9
PNScan	87.7	6.0	6.3	89.8	5.4	4.8	94.8	3.1	2.1	0.93	7.1	7.0
Average	**79.3**	**10.6**	**10.1**	**81.9**	**7.7**	**10.4**	**88.9**	**5.2**	**5.9**	**0.88**	**9.7**	**7.7**

passwords. After being infected by this malware, the device becomes a botnet robot and launches a high-intensity botnet attack under the command of a hacker.

Our benign programs include system binaries such as *ping* and *netstat*[1]. The traced values gathered by running both malware and benign programs on the hardware board are utilized as inputs to our classifier, as described next.

C. Data Acquisition

As discussed in Section III-A, we have collected data through the following three avenues.

Hardware Performance Counters (HPC): To gather HPC data, we make a serial connection to the target device and, and run both the malicious and benign programs with the *perf* record command. Within the perf command, we set various HPC values (such as branch mispredictions, number of branches, number of loads, etc.) as traced events.

Embedded Trace Buffer (ETB): To obtain ETB traces, we run malicious programs on the target system while connected to Xilinx SDK through Ethernet. Within Xilinx SDK, we add breakpoints to various code locations and dump trace data at these breakpoints. Our studies have shown that tracing register values provide better insight compared to tracing other signals.

Synergistic Combination of HPC and ETB: As shown in Figure 2, we dump ETB data only when our ML models find any suspicious HPC values. To associate regions of HPC values with ETB values, we add dynamic tracepoints (using the perf probe functionality) to various functions in the malicious programs. The dynamic tracepoints traced during HPC data generation allow us to associate regions of HPC data with ETB data. By remembering which function was called near a suspicious HPC data region, we can place a breakpoint in the ETB region associated with suspicious HPC activity. In other words, we can generate ETB data corresponding to specific HPC values.

D. Effectiveness of Data Compaction

Figure 7 shows that the data compaction is effective for both decision tree and LSTM-based RNN models. We train both ML models with the same hyperparameter and training set. The only difference is that one model is further refined by data compaction step (tree Pruning for DT and outlier elimination for LSTM) as mentioned in Section III-C. As we can see, there is a huge difference in stability. Model with data compaction technique demonstrates stable performance across 100 trials. However, there are drastic variations in accuracy for the models without data compaction. When applied to decision tree, this phenomenon becomes more obvious, where high accuracy (e.g., 90%) was obtained occasionally, but the accuracy can also drop below 70%. In reality, a stable performance is desirable, otherwise a user needs to try numerous times to obtain an acceptable result, which can be infeasible for large scale tasks. In the following sections, our 'proposed approach' refers to our proposed approach with data compaction techniques.

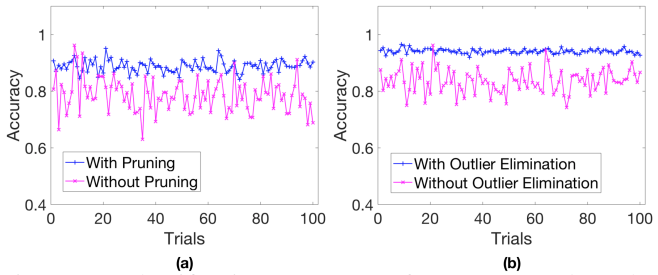


Figure 7: Classification accuracy of our proposed method with/without data compaction. (a) DT-based methods with/without tree pruning. (b) LSTM-based methods with/without outlier elimination.

E. Impact of Data Acquisition Methods

We have evaluated the effectiveness of three data acquisition methods (HPC, ETB and HPC+ETB) using two machine learning models (DT and LSTM). The performance of proposed method (implemented in DT) is presented in Table I. It compares malware detection accuracy, false positive (FP) and false negative (FN) using HPC, ETB as well as combination (HPC+ETB) of them. From a global perspective, the detection accuracy is decreased in many aspects under this configuration compared with that using LSTM (Table II). Especially for Mirai family, with data collected from HPC, the detection accuracy by DT is merely 69.8%, which is unacceptable for real applications. As expected, the decision tree is not suitable for detecting malware with complex behaviors. This is an inherent problem since decision tree assumes that the pattern of malicious behavior follows a straight line of sequential logic, where the decision is made based on a sequence of feature checking. Since Mirai and Bashlite works in a client-server mode, there are many benign function calls, and the working pattern is very complex. Interestingly, decision tree works much better in PNScan dataset, since the mechanism behind PNScan is breaking down users' secret password in a brute-force manner, the behavior is easier to detect and classified by a decision tree.

Therefore, by interpreting the ML model, we clearly have a guideline on how DT classifies PNScan malware. However, this interpretation also indicates crucial problems for DT-based solutions. As we observe from the above example, the DT

[1]ping and netstat are important since our malware are botnets.

Table II: Comparison of detection accuracy with different acquisition approaches for RNN (LSTM)

	HPC			ETB			HPC+ETB					
Benchmarks	**Accuracy** (%)	**FP** (%)	**FN** (%)	**Accuracy** (%)	**FP** (%)	**FN** (%)	**Accuracy** (%)	**FP** (%)	**FN** (%)	**F1 Score**	**improv./HPC** (%)	**improv./ETB** (%)
Bashlite	89.1	3.7	7.2	94.1	1.3	4.6	100.0	0.0	0.0	1.0	7.1	5.9
Mirai	72.6	12.9	14.5	81.2	11.1	7.7	93.3	2.9	3.8	0.93	20.7	12.1
PNScan	86.1	11.0	2.9	96.2	2.3	1.5	99.8	0.0	0.2	0.98	13.7	3.6
Average	**82.6**	**9.2**	**8.2**	**90.5**	**4.9**	**4.6**	**97.7**	**0.9**	**1.4**	**0.97**	**15.1**	**7.2**

model makes prediction barely based on behavioral patterns of program instead of the intention of certain behaviors. Therefore, it is easy to mispredict benign programs with similar behavior pattern as malware. We recognize this as the root cause for high false positive among all DT-based solutions in Table I. Table I and Table II show that ETB data leads to better accuracy compared to HPC values for both ML models. In fact, an effective combination of HPC and ETB provides the best performance compared to HPC or ETB alone. In the following sections, we use the combined data (HPC+ETB) when we compare with related approaches.

F. Malware Detection Performance

Figure 8 compares our proposed methods with the state-of-the-art (PREEMPT [3]). We have provided the performance across multiple trials since we consider stability as one of the important attributes for malware detection. We have also provided the average values for easier discussion and improved readability (the numbers in the legend box provide the average values for the respective methods). PREEMPT utilizes embedded trace buffer to help reducing latency and overcoming malware equipped with obfuscation. PREEMPT utilizes two types of implementation, random forest (PREEMPT_RF) and decision tree (PREEMPT_DT). Note that both of our methods (Proposed_DT and Proposed_LSTM) use the combined data (HPC+ETB).

To obtain these results, we ran both malicious and benign software on our hardware platform. We executed a total of 367 programs (including both malicious and benign ones) and all the traced data were mixed up and further split into training (80%) and test (20%) sets after labeling. Total training epochs are 200 for every model and we plot test accuracy every 10 epochs. To fulfill fair comparison, the threshold for classificatiion is setup to 0.9 for all configurations. The performance of all methods are depicted in Figure 8.

Figure 8 compares the prediction accuracy of our approach with PREEMPT_RF and PREEMPT_DT. As we can see, our proposed method provides the best malware detection accuracy. For Bashlite, our proposed method achieved 89% accuracy with DT and near 100% with LSTM, while PREEMPT attained a maximum accuracy of 86.5% with RF. For PNScan, both proposed method and Preempt_RF performed well and the best accuracy our method achieved is 98.8%. The PREEMPT appeared fragile in the face of Mirai, with an average of 62.7% accuracy for DT and 81.9% for RF, while the proposed method using LSTM model provided an average accuracy as high as 94%. Note the inferior performance of PREEMPT_DT in Mirai dataset, which clearly shows the lack of PREEMPT's ability to handle complex malware.

Meanwhile, we provide *Receiver operating characteristic* (ROC) curves for better evaluating the classifier output in Figure 9, and the *Area under the Curve* values are presented in Table III.

Table III: Area Under the Curves (AUC)

Methods \ **Malware**	**BASHLITE**	**PNScan**	**Mirai**
Proposed_DT	0.9397	0.9694	0.9446
Proposed_LSTM	0.9836	0.9941	0.9801
PREEMPT_RF	0.9244	0.9614	0.9411
PREEMPT_DT	0.8822	0.9212	0.7859

If we omit malware and test models on traced data gathered from benign software only, Figure 8(d) shows the false positive rate (FPR) of all four methods. The diagram illustrates the major drawback of PREEMPT and DT-based approaches, it possesses an average FPR as high as 20.9% with PREEMPT_RF, 31.6% with PREEMPT_DT, and 16.4% with Proposed_DT. In other words, they are very likely to mispredict a benign software as malware. Tested benign software samples also executes Linux system binaries like *netstat* and *ping*, which are also frequently executed by botnet malware. Since the above three methods cannot analyze time sequential data, they failed to recognize benign execution of these binaries with the help of context and produced wrong predictions. In contrast, our LSTM-based framework obtained FPR as low as 3.4%.

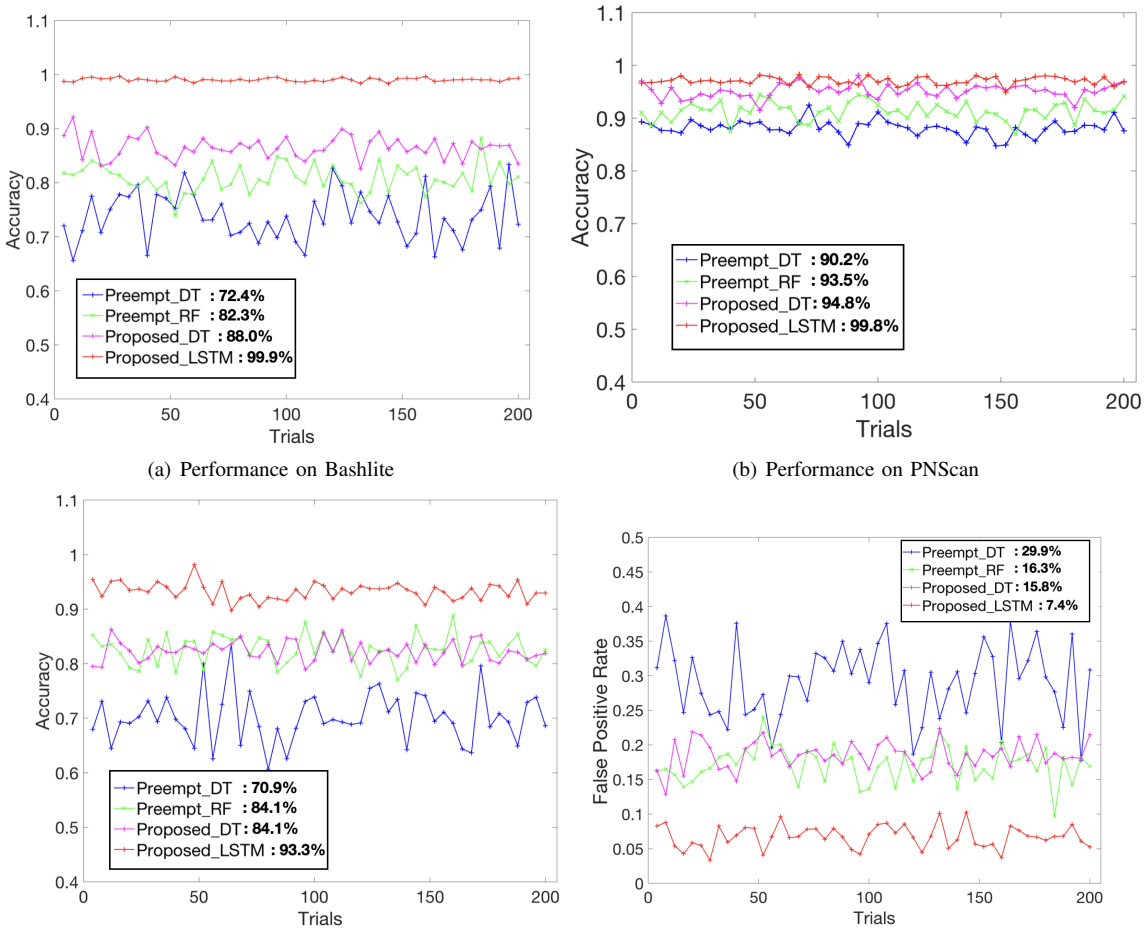
G. Comparison of Time Efficiency

The results reported so far provide us a comprehensive view of accuracy performance of all methods. One observation is LSTM-based methods are obviously better than DT-based methods. This difference is caused by DT's sequential selection logic. However, when taking model training and testing time into consideration, there is a trade-off between detection accuracy and time efficiency, as depicted in Table IV.

Table IV: Comparison of time efficiency

Model	Data Source	Training (second)	Testing (second)	Total (second)
DT	HPC	17.9	2.7	20.6
	ETB	34.6	4.1	38.7
	HPC+ETB	30.1	4.5	34.6
LSTM	HPC	104.5	44.3	148.8
	ETB	226.3	55.4	281.7
	HPC+ETB	187.1	63.0	250.1

We have explored various models for 200 trials and we report the average training, testing and total time cost. As we can see, DT-based methods runs much faster (almost an order-of-magnitude) than its LSTM-based counterpart. Especially when it comes to the testing time, DT's simple structure determines its low execution time cost, while LSTM requires a full



(a) Performance on Bashlite

(b) Performance on PNScan

(c) Performance on Mirai

(d) False Positive Rate (FPR)

Figure 8: Performance of machine learning models: (a) - (c) for various malware, and (d) for benign benchmarks. The numbers in the legend box provides the average values for the four methods.

forward pass of the neural network, which requires hundreds of multiplication and summation operations. Therefore, DT-based methods can be used as a quick (fast) check in practice, while LSTM-based approach is suitable for deep (slow) scan.

H. Evaluation of Explainability

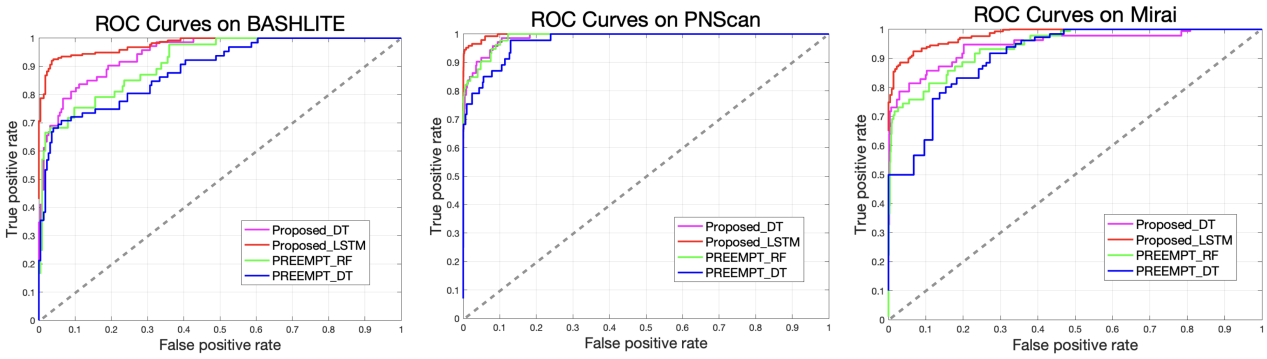
Explainability is utilized for interpreting classification results. In this section, we highlight five major advantages of explainability for improving malware detection performance.

Adversarial Training: Explainable ML is beneficial for adversarial training to protect from adversarial attacks. When a user encounters false positive or false negative outcome, it tells user what characteristics are likely to cause incorrect prediction. Malicious samples with similar characteristics can be synthesized and merged into the pool of training set to retrain the ML model, thereby enhancing the robustness of the model against known attacks. Section IV-I discusses this topic in details.

Additional Information for Malware Localization: Explainable ML also provides additional information during malware detection. Specifically, the top features with large coefficients are likely to be related to the malicious behavior. We can also check the clock cycle distribution of these top features to gather information about the malware. Figure 11 and Figure 12 provide two illustrative examples.

Utilization of Incorrect Results: Existing malware detection works cannot reach 100% accuracy in all cases. Traditionally, there is no standard way to tackle incorrect prediction results, and the common solution is to re-train the model with those mispredicted samples to enforce the model remembering them. It treats the symptoms but not the root cause. It remains unclear what new features are learned from the re-training so that the model can correct its mistakes. The model merely remembers the name, size, or other aspects of the sample, instead of actually recognizing the reason for previous mistakes. For example, when a new type of malware comes with similar intrinsic but different superfluous features, the model is likely to make mistake again. In contrast, explainable ML can point out the reason for its prediction. In other words, the user can tell why the misprediction happened (e.g., which feature is the dominant reason for this mistake). Therefore, when retraining the model, synthetic samples which emphasize that particular feature can be crafted for improved accuracy.

Flexibility in Malware Detection: Explainable ML significantly improves the flexibility of malware detection method. The limitation of non-explainable ML method is its flexibility, especially for unseen malware. To continuously upgrade itself, it usually relies on sufficient iterations of training with samples of newly found malware. In general, there may not



(a) BASHLITE (b) PNScan (c) Mirai

Figure 9: Receiver operating characteristic curves of different methods against various malware.

be sufficient number of samples. With the help of explainable ML, synthetic samples with similar attributes can be generated to serve as training samples. In other words, ML models with explainability can better fix itself for unseen benchmarks/malware types.

Iterative Refinement: Although classification (traditional ML) and explainable ML runs in parallel, the additional information generated by the current iteration of explainable ML (e.g., the terms with larger coefficients) is helpful for the classification in the next iteration. This symbiotic relationship works in a way where explainable ML is performed by interpreting the outcome of the trained ML model, while the interpretation can be utilized in the next iteration of model training. In other words, explainable ML can play the role of an assistant in the process of model training.

The remainder of this section provides three illustrative examples of utilizing explainability for malware detection and localization. Figure 10 provides an example of how DT-based method works on detecting a PNScan attack.

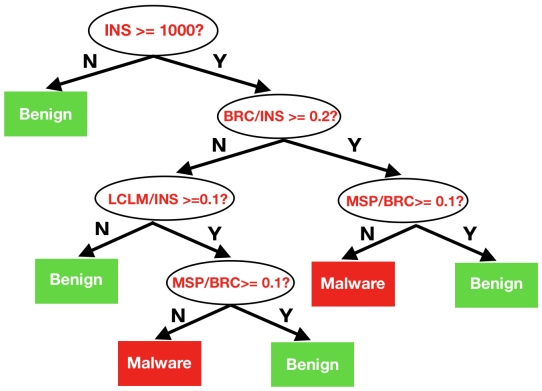


Figure 10: Interpretation of DT model for detecting PNScan

As we can see, the key hardware events selected are:

- INS: Total number of instructions.
- BRC: Total number of branch instructions.
- MSP: Total branch mispredictions.
- LCLM: Total L1-cache-load-misses

The tree traversal begins with evaluating the total number of instructions to check if the program size satisfies commonly met PNScan malware. Next, it measures two fundamental properties of the program, the fraction of branch instructions and cache load misses. This is due to the fact that PNScan performs dictionary-based attacks in attempts to brute-force getting info about victim network. Therefore, the DT measures the branch instructions and mispredictions to check if target programs satisfy this brute-force behavior. Since PNScan is dictionary-based attack, even if the DT model fails to detect malware based on branch instructions, it further checks the statistics about cache misses to see if the program heavily requires memory-access of a fixed address.

	C_1	C_2	C_3	C_4	C_5	C_6	C_7	...
R_1	8ca0	bc00	1886	8ca0	a68c	c401	e401	
R_2	a2c2	08b4	a6ab	e6a2	0004	88fd	b422	
R_3	b485	ec00	28d4	2101	506c	02f0	02e2	
R_4	8c21	6002	d201	90c4	02f9	90a2	0048	
...								
	0.003	0.028	0.073	0.431	0.136	0.288	0.001	

Figure 11: Interpretation of BASHLITE client's traced signals

Figure 11 shows an example of detecting Bashlite's client on host machine using RNN (LSTM)-based approach. Specifically, it shows a snapshot of the trace table, where each row represents the values in a register in specific clock cycles (each column represents a specific clock cycle). In this example, we computed the corresponding contribution factor of each clock cycle towards the RNN output using linear regression, which is shown as weights in the last (colored) row. As we can see, the weight of C_4 is significantly larger than the others. This immediately indicates the clock cycle of malicious behavior. By tracing the execution, we find that C_4 points to the timestamp before the start of function "*processCmd*" in Bashlite, which is the most important function of Bashlite to perform its malicious functionality. In other words, this is the starting point and exact reason for recognizing this program as malware. it enables the client to process the commands received from the server to decide the next moves. This is always the beginning of the client to perform malicious behavior and the exact reason for recognizing it as malware.

Another example of outcome interpretation is shown in Figure 12, where we measure the contribution of each traced register signal. The given data is the trace table of executing Mirai's bot on host machine. This time we evaluate the contribution row-by-row, and the result is listed on the right

Table V: The impact of data compaction against different adversarial attacks

Models / Attacks	Compacted_DT			Compacted_LSTM			Uncompacted_DT			Uncompacted_LSTM		
	Bashlite	**Mirai**	**PNScan**	**Bashlite**	**Mirai**	**PNScan**	**Bashlite**	**Mirai**	**PNScan**	**Bashlite**	**Mirai**	**PNScan**
N/A	88.0	84.1	94.8	100.0	93.3	99.8	79.3	76.5	69.4	82.1	76.0	75.9
FGSM(ε = .1)	53.3	62.8	54.5	71.2	51.1	47.7	70.3	62.9	63.8	70.9	55.7	60.1
JSMA ($\theta, \gamma = .1, 1$)	56.2	46.7	65.1	68.6	58.9	44.4	77.0	66.6	68.2	80.2	71.3	59.5
DeepFool($\varepsilon = 1e-6$)	47.1	51.6	62.9	46.2	42.3	41.5	53.8	50.0	72.2	49.9	63.7	43.6
PGD(($\varepsilon = 0.3$))	0.4	11.0	2.9	6.2	22.3	11.5	19.8	20.0	22.2	33.9	53.7	23.6
Average	**49.0**	**51.4**	**56.1**	**58.4**	**53.6**	**48.9**	**60.4**	**55.2**	**59.1**	**63.4**	**64.1**	**52.5**

Table VI: Detection accuracy against different adversarial attacks after applying defence

Models / Attacks	Unprotected_DT			Unprotected_LSTM			Protected_DT			Protected_LSTM		
	Bashlite	**Mirai**	**PNScan**	**Bashlite**	**Mirai**	**PNScan**	**Bashlite**	**Mirai**	**PNScan**	**Bashlite**	**Mirai**	**PNScan**
N/A	88.0	84.1	94.8	100.0	93.3	99.8	89.8	86.1	95.0	100.0	93.0	99.9
FGSM(ε = .1)	53.3	62.8	54.5	71.2	51.1	47.7	92.0	82.4	89.3	99.9	85.7	97.2
JSMA ($\theta, \gamma = .1, 1$)	56.2	46.7	65.1	68.6	58.9	44.4	82.4	86.1	87.1	98.2	91.2	95.9
DeepFool($\varepsilon = 1e-6$)	47.1	51.6	62.9	46.2	42.3	41.5	83.8	70.0	62.2	79.9	83.0	92.2
PGD(($\varepsilon = 0.3$))	0.4	11.0	2.9	6.2	22.3	11.5	25.5	16.0	32.2	33.9	43.0	53.4
Average	**49.0**	**51.4**	**56.1**	**58.4**	**53.6**	**48.9**	**74.7**	**68.2**	**73.1**	**82.4**	**79.1**	**87.7**

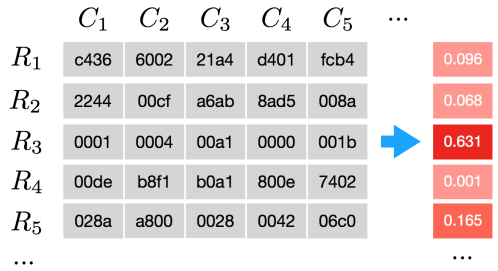


Figure 12: Interpretation of Mirai bot's traced signals

side of the trace table. As we can see, register R_3 is recognized as the most important factor. Here R_3 is storing the variable "*ATTACK_VECTOR*" in Mirai. This variable records the identity of attack modes, based on which the bot takes relative actions to perform either a UDP attack or DNS attack. This attack-mode flag is the most important feature of a majority of malware bot programs, and our proposed method successfully extracted it from the traces to illustrate the reason for making this prediction.

I. Evaluation of Robustness

An ML model's robustness against adversarial attacks is an important consideration. Specifically, in our proposed method, the data compaction step filters out the least important features. The remaining features with high weight values indicate that a small change in the input will lead to a drastic change in the output. From the perspective of an adversary, these are the preferred features to conduct adversarial attacks since small changes to the program can change the final outcome. We have explored the available state-of-the-art adversarial attacks, and selected the following four attack algorithms for evaluation of robustness.

- FGSM [34]: A gradient based lightweight attack algorithm. Usually applied for sanity check.
- DeepFool [35]: An untargeted attack technique optimized for the $L2$ distance metric.
- JSMA [36]: A Jacobian-based Saliency Map attack.
- PGD [37]: Projected gradient descent based attack. It is among the strongest attacks utilizing the local first order information of input values.

Table V shows the evaluation results. It also provides the fine-tuned hyperparameters of each adversarial attack method. As expected, due to the selection of filtered features, ML models trained with compacted dataset perform slightly worse against adversarial attacks. To mitigate this challenge, we have applied the following two strategies to enhance the robustness of ML models.

1) *Adversarial Training:* We have utilized the traditional adversarial training, where adversarial samples were crafted and mixed up in the pool. A rigorous training with such a sample pool enforces the ML models to distinguish adversarial samples and normal samples.
2) *Spectral Normalization:* We have also utilized a strategy called 'spectral normalization' proposed in [38] to further reduce the ML model's sensitivity to obfuscations.

The effect of applying the proposed defense strategy is demonstrated in Table VI. The results demonstrate that our defensive algorithms indeed help protecting models from adversarial attacks. As we can see, our proposed method improved by defence strategies provides decent robustness while the unprotected method appear fragile in the face of adversarial attacks. For lightweight attacks such as gradient-based ones (like FGSM), the proposed approach is almost unaffected. Only for powerful attacks like PGD, there is a significant drop in performance.

J. Evaluation of Overhead

To evaluate the practicality of the proposed method, we optimized the implementation of the detection hardware and evaluated hardware overhead in the following two ways and reported the summary of the observations. We have implemented our ML model in Verilog and performed area and power estimation using Vivado on Xilinx Zynq-7000 SoC ZC702 board. To optimize the implementation, we have applied the input dataset in multiple iterations instead of applying all of them once. The power consumption is only few Watts. In terms of area, it requires 19,965 LUTs (out of 53,200 LUTs with an utilization of 38%). We have also used Intel Power Gadget to perform power analysis at the system level while trying to

detect a specific malware. Table VII shows the total power consumption of the processor core and DRAM. The average power consumption is few Watts. As we can see, our optimized implementation leads to minor area and power overhead.

Table VII: Power consumption using Intel Power Gadget

Malware	Bashlite	PNScan	Mirai	Average
Power (W)	5.34	5.51	3.75	4.87

K. HPC Reliability Concerns

Recent studies [21], [22] have highlighted serious reliability concerns for HPC-based malware detection. In this section, we discuss why our proposed approach does not have these limitations. In [21], the author highlighted five critical challenges in existing works for HPC-based malware detection. Our proposed framework addressed these challenges as follows.

- *Dynamic Binary Instrumentation (DBI):* In our experimental setup, we did not utilize dynamic binary instrumentation tools to extract HPC values but used *perf* instead. Therefore, our approach does not inherit any limitations associated with DBI.
- *Virtual Machines (VMs):* We have utilized Zynq SoC hardware board instead of VMs for evaluation. Therefore, VM-related concerns are not applicable for our approach.
- *Insufficient Validations:* We have performed sufficient validation as discussed in previous sections.
- *Few Data Samples:* We have explored three popular malware families and collected data from diverse sources including embedded trace buffer as well hardware performance counter. The github repository describing the experimental setup as well as the dataset is available at *https://github.com/Jshel/MalwareDetection*.

In [22], the author outlined five critical pitfalls to avoid in utilizing HPC for security tasks. Our work also addressed them as follows.

- *Profiling Tools:* We did not compare performance based on HPC with those from other profiling tools (e.g., Pin).
- *CPU Architectures:* We specifically discussed the CPU for conducting our experimental evaluation in Section IV-A.
- *Preprocess Filtering:* We have performed this step as discussed in the data compaction step in Section III-C.
- *Adversary:* Section IV-I provides the analysis of our proposed method against adversarial attacks.
- *Documenting Work:* We have a created a repository describing the experimental setup and data traces at *https://github.com/Jshel/MalwareJournalExtension*. Researchers can use these vast dataset to reproduce our experimental results as well as to perform further research in this field.

V. Conclusion

Malicious software (malware) is a serious threat to modern computing systems. Existing software based solutions are not effective in the face of attacks with obfuscation or other evasive techniques. Recent hardware-based detection techniques are promising but their detection accuracy can still be improved. Moreover, the classification results cannot be interpreted in a meaningful way. Our proposed approach addresses these limitations by developing a regression-based explainable machine learning algorithm. In this paper, we explore various machine learning models to make fast decisions using hardware performance counters as well as embedded trace buffer. Our approach is able to find the major contributors among all input features to help interpret the classification results, which is utilized to either support correct predictions, or justify mispredictions through adversarial training. Experimental results demonstrated that our approach significantly outperforms (97.7% accuracy on average) state-of-the-art approaches on several benchmarks, and provides explainable interpretation on detection results at the same time.

Acknowledgments

This work was partially supported by the National Science Foundation (NSF) grant CCF-1908131.

References

[1] Kelly Bissell and Larry Ponemon. The cost of cybercrime. https://www.accenture.com/t20190305T185301Z__w__/us-en/_acnmedia/PDF-96/Accenture-2019-Cost-of-Cybercrime-Study-Final.pdf#zoom=50, 2019.

[2] Zahra Bazrafshan et al. A survey on heuristic malware detection techniques. In *ICIKF*, pages 113–120, 2013.

[3] Kanad Basu et al. PREEMPT: preempting malware by examining embedded processor traces. In *DAC*, page 166, 2019.

[4] Xueyang Wang et al. Hardware performance counter-based malware identification and detection with adaptive compressive sensing. *ACM TACO*, 13(1):3, 2016.

[5] Yann LeCun, Yoshua Bengio, and Geoffrey E. Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[6] Daniel Arp et al. Effective and efficient malware detection at the end host. In *18th USENIX*, Montreal, Quebec, 2009.

[7] George E. Dahl, Jack W. Stokes, Li Deng, and Dong Yu. Large-scale malware classification using random projections and neural networks. In *ICASSP*, pages 3422–3426, 2013.

[8] Kathrin Grosse et al. Adversarial perturbations against deep neural networks for malware classification. *CoRR*, 2016.

[9] Joshua Saxe and Konstantin Berlin. Deep neural network based malware detection using two dimensional binary program features. In *10th MALCON*, pages 11–20, 2015.

[10] Qinglong Wang et al. Adversary resistant deep neural networks with an application to malware detection. In *Proceedings of the 23rd ACM SIGKDD*, pages 1145–1153, 2017.

[11] Nwokedi Idika and Aditya Mathur. A survey of malware detection techniques. *Purdue University*, 03 2007.

[12] A. Moser, C. Kruegel, and E. Kirda. Limits of static analysis for malware detection. pages 421–430, Dec 2007.

[13] *Malware Obfuscation Techniques: A Brief Survey*. IEEE Computer Society, 2010.

[14] Grégoire Jacob, Hervé Debar, and Eric Filiol. Behavioral detection of malware: from a survey towards an established taxonomy. *Journal in Computer Virology*, 4(3):251–266, 2008.

[15] Suman Jana and Vitaly Shmatikov. Abusing file processing in malware detectors for fun and profit. In *IEEE S&P*, 2012.

[16] Nick L. Petroni Jr. et al. Copilot - a coprocessor-based kernel runtime integrity monitor. In *Proceedings of the 13th USENIX Security Symposium*, pages 179–194, 2004.

[17] John Demme et al. On the feasibility of online malware detection with performance counters. In *The 40th Annual ISCA*, pages 559–570, 2013.

[18] Mikhail Kazdagli et al. Quantifying and improving the efficiency of hardware-based mobile malware detectors. In *49th Annual IEEE/ACM MICRO*, pages 37:1–37:13, 2016.

[19] Xueyang Wang and Ramesh Karri. Numchecker: detecting kernel control-flow modifying rootkits by using hardware performance counters. In *DAC*, pages 79:1–79:7, 2013.

[20] Zhixin Pan, Jennifer Sheldon, Chamika Sudusinghe, Subodha Charles, and Prabhat Mishra. Hardware-assisted malware detection using machine learning. In *Design Automation and Test in Europe (DATE)*, 2021.

[21] Boyou Zhou, Anmol Gupta, Rasoul Jahanshahi, Manuel Egele, and Ajay Joshi. Hardware performance counters can detect malware: Myth or fact? In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, pages 457–468, 2018.

[22] Sanjeev Das, Jan Werner, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. Sok: The challenges, pitfalls, and perils of using hardware performance counters for security. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 20–38. IEEE, 2019.

[23] Zhixin Pan, Jennifer Sheldon, and Prabhat Mishra. Hardware-assisted malware detection using explainable machine learning. In *2020 IEEE 38th International Conference on Computer Design (ICCD)*, pages 663–666. IEEE, 2020.

[24] Zhixin Pan and Prabhat Mishra. Hardware acceleration of explainable machine learning. In *Design Automation and Test in Europe (DATE)*, 2022.

[25] Prabhat Mishra, Ronny Morad, Avi Ziv, and Sandip Ray. Post-silicon validation in the soc era: A tutorial introduction. *IEEE Design & Test*, 34(3):68–92, 2017.

[26] Prabhat Mishra and Farimah Farahmandi. *Post-Silicon Validation and Debug*. Springer, 2019.

[27] J. Ross Quinlan. Induction of decision trees. *Mach. Learn.*, 1(1):81–106, 1986.

[28] Sepp Hochreiter. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 6(2):107–116, 1998.

[29] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. Understanding the exploding gradient problem. *CoRR*, 2012.

[30] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986.

[31] Hasim Sak et al. Long short-term memory based recurrent neural network architectures for large vocabulary speech recognition. *CoRR*, abs/1402.1128, 2014.

[32] Junyoung Chung et al. Empirical evaluation of gated recurrent neural networks on sequence modeling. *CoRR*, 2014.

[33] Kishore Angrishi. Turning internet of things(iot) into internet of vulnerabilities (iov). *CoRR*, 2017.

[34] Ian Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. pages 1–10, 01 2015.

[35] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, and Pascal Frossard. Deepfool: a simple and accurate method to fool deep neural networks. *CVPR*, 11 2016.

[36] Rey Wiyatno and Anqi Xu. Maximal jacobian-based saliency map attack. *CoRR*, abs/1808.07945, 2018.

[37] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018.

[38] Zhixin Pan and Prabhat Mishra. Accelerating spectral normalization for enhancing robustness of deep neural networks. In *2021 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 20–38. IEEE, 2021.

Zhixin Pan is a Ph.D student in the Department of Computer & Information Science & Engineering at the University of Florida. He received his B.E. in the Department of Software Engineering from Huazhong University of Science & Technology, Wuhan, China in 2015. His area of research includes Cyber & Hardware Security, post-silicon debug, data mining and machine learning.

Jennifer Sheldon is a Ph.D student in the Department of Computer & Information Science & Engineering at the University of Florida. She received her B.S. in Computer Engineering from University of Florida in 2020. Her area of research includes side-channel analysis, hardware security and trust, and malware detection.

Prabhat Mishra is a Professor in the Department of Computer and Information Science and Engineering at the University of Florida. He received his Ph.D. in Computer Science from the University of California at Irvine in 2004. His research interests include embedded and cyber-physical systems, hardware security and trust, and energy-aware computing. He currently serves as an Associate Editor of IEEE Transactions on VLSI Systems and ACM Transactions on Embedded Computing Systems. He is an IEEE Fellow and an ACM Distinguished Scientist.