An Effective Approach for Malware Detection and Explanation via Deep Learning Analysis

Huozhu Wang
Institute of Information Engineering,Chinese Academy of Science
School of Cyber Security, University of Chinese Academy of Sciences
Beijing, China
wanghuozhu@iie.ac.cn

Ziyuan Zhu*
Institute of Information Engineering,Chinese Academy of Science
School of Cyber Security, University of Chinese Academy of Sciences
Beijing, China
zhuziyuan@iie.ac.cn

Zhongkai Tong
Institute of Information Engineering,Chinese Academy of Science
School of Cyber Security, University of Chinese Academy of Sciences
Beijing, China
tongzhongkai@iie.ac.cn

Xiang Yin
Institute of Information Engineering,Chinese Academy of Science
School of Cyber Security, University of Chinese Academy of Sciences
Beijing, China
yinxiang@iie.ac.cn

Yusi Feng
Institute of Information Engineering,Chinese Academy of Science
School of Cyber Security, University of Chinese Academy of Sciences
Beijing, China
fengyusi@iie.ac.cn

Gang Shi
Institute of Information Engineering,Chinese Academy of Science
School of Cyber Security, University of Chinese Academy of Sciences
Beijing, China
shigang@iie.ac.cn

Dan Meng
Institute of Information Engineering,Chinese Academy of Science
School of Cyber Security, University of Chinese Academy of Sciences
Beijing, China
mengdan@iie.ac.cn

2021 International Joint Conference on Neural Networks (IJCNN) | 978-1-6654-3900-8/21/$31.00 ©2021 IEEE | DOI: 10.1109/IJCNN52387.2021.9534115

***Abstract*—The next generation attackers often generate malware variants with Artificial Intelligence (AI) weapons, which are deliberately designed to evade antivirus engines. Security defenders propose many AI-based approaches to detect the massive number of malware variants. However, most AI-based malware detection approaches only output a label to users, and these labels are mainly unexplainable. The lack of transparency has introduced many black-box attacks. Malware developers can develop adversarial examples to evade these AI-based malware detection systems. In this paper, we propose an effective approach for malware detection and explanation, which can locate malicious code snippets by explaining the malware classifier decision result. To this end, firstly, we get the system call number sequence of the target sample with instrumentation tools in an elaborated sandbox. Secondly, we feed the mapped system call number sequence into a deep learning model to make a decision on whether the target sample is benign or malicious. Thirdly, we adopt the Layer-wise Relevance Propagation algorithm to find which slice of a sequence makes the greatest contribution in the decision. Our evaluation demonstrates that our approach achieves high classification accuracy (97.39%), reduces the neural network size by 20 times, and saves the malware analyst time to locate malicious code snippets.**

Keywords—Artificial Intelligence, Cybersecurity, Malware detection, Recurrent Neural Network, Interpretability.

I. INTRODUCTION

With the rapid development of wireless networks and edge computing, malware in the edge node introduced a lot of serious security and privacy issues [1]. From the view of the internal attack surface, there are a growing number of protocol weaknesses and vulnerabilities in software and hardware systems [2]. From the view of the external attack surface, malware is the first point of initiation for external attacks.

Based on the AV-TEST Institute report [3], the AV-TEST Institute registers over 350 thousand new malicious programs (malware) and potentially unwanted applications (PUA) every day [3]. Although the last decade has introduced many malware detection approaches, malware developers usually create new malware variants with dedicated anti-analysis features to evade these detection approaches [4].

One challenge is that the next generation attackers often generate malware variants with Artificial Intelligence (AI) weapons to disable automated static and dynamic malware analysis systems [5]. On the one hand, to evade static malware analysis systems, malware variants have been developed with AI weapons [6]. Some adversaries learn to evade detection based on changing static features of executable files via reinforcement learning [7]. Some adversaries generate adversarial examples to disable CFG-based malware classifiers [8]. On the other hand, to evade dynamic malware analysis systems, malware often automatically detects dynamic analysis system fingerprints to behave benignly with AI weapons. These fingerprints include username, IP, files, drivers, system settings, virtualized environments, and running processes, etc. Moreover, some trigger-based malware needs special conditions (e.g., timing, input, or network) to trigger its malicious behavior [5].

For tackling the AI weapon, many AI-based methods with API call sequences [9] have been proposed to detect a large number of new malware variants. These detection approaches obtain a higher accuracy than traditional signature-based methods. But challenges still exist. Firstly, such API call sequence-based approaches cannot capture malware behavior,

* Corresponding author: Ziyuan Zhu, zhuziyuan@iie.ac.cn.

978-0-7381-3366-9/21/$31.00 ©2021 IEEE

which is programmed with inline assembly. So these traditional approaches will have a high false-negative rate when there is malware programmed with inline assembly in the dataset. For example, in [9], Amer et al. list many malware detection approaches which used API call, but these methods missed the system call in the form of inline assembly [47]. Secondly, these neural network models, such as CNN or RNN, usually cannot explain the classifier result in malware detection. Security defenders cannot understand why the sample is malware. Attackers can develop many adversarial examples to evade these Neural Network-based malware detection systems. Therefore, we propose an effective approach for malware detection and explanation via deep learning analysis, including a classifier and an interpreter. The classifier predicts whether the sample is malicious, and the interpreter explains the classifier result via a system call number sequence of the target sample. The experimental results show that our approach can help the security community to locate key malicious code snippets.

Contribution: In summary, our contributions break down into the following aspects:

- We propose an effective approach for malware detection via deep learning analysis. The approach just needs a small amount of feature data (system call number sequence) and obtains a high classifier accuracy (97.39%) for malware detection.
- To the best of our knowledge, this paper is the first paper to use system call number sequence in malware detection. The feature brings more advantages than API call in previous work, and the most significant advantage is that our approach can capture malicious behavior of malware programmed with inline assembly.
- We explain the malware classification results generated by our deep learning model. The explanation is system call number sequence, which can help security analysts replay the malware execution path. And even help the security community locate key malicious code snippets automatically.

Road map: The rest of the paper is organized as follows. In section II, we discuss an overview of related works. Section III describes our proposed model and algorithm. Section IV elaborates our dataset and experimental setting details. We present detailed results on the evaluations of our proposed approach in section V. Conclusion are presented in section VI.

II. RELATED WORKS

This section mainly focuses on two parts: malware detection and explanation.

A. Malware Detection Methods

According to whether we execute the target program sample or not, malware detection approaches can be mainly divided into static detection and dynamic detection.

In static detection methods, we can extract some static features from the target programs, which include strings, opcodes, API calls, byte n-grams, and control flow graphs [12]. Kirat et al. [13] proposed an automated technique for extracting evasive malware signatures via searching evasive behavior in system call sequences. Because malware developers usually hide small malicious code snippets in benign code by using some advanced techniques such as obfuscation, dynamic code loading, etc. [11], these static detection approaches usually cannot find the key malicious code snippet.

Unlike static detection approaches, dynamic detection approaches monitor the target program in a virtual execution environment or bare metal system [5]. Some deliberately designed malware usually detects the virtual execution environment (e.g., emulator, hosted-VM, and hypervisor) by executing red pills instructions [15] to behave benignly. So researchers turned to analyze malware on bare metal. Kirat et al. [16] proposed an evasive malware detection approach by comparing the malware behavior monitored in the bare-metal system with other popular malware analysis systems. Because some rootkits have the ability to change the normal behaviors of the operating system, many studies turn to focus on malware features extracted from the processor. Ozsoy et al. [17] embed a logistic regression model into the processor for real-time hardware-based malware detection. Xu et al. [27] proposed an online approach for detecting malware based on the target program's virtual memory access pattern. Sayadi et al. [28] collected micro-architectural events such as processor Hardware Performance Counters (HPCs) data for malware detection at run-time. Subsequent efforts like Sehatbakhsh et al. [18] detected malware in real-time by externally monitoring electromagnetic signals emitted by an electronic hardware device (e.g., a microprocessor). We can notice that extracted features are from software to hardware to against evasive malware.

Static and dynamic malware detection approaches usually extract API calls of the target executable file. Then use these extracted features to train a Neural Network-based malware classifier. However, such API call sequence-based approaches [10, 12, 13, 14, 15, 16] usually hook API calls. But cannot capture malicious behaviors of malware that are programmed with inline assembly. We show two forms of a system call (sys_close) in Fig. 1. Previous API call sequence-based approaches will miss some malware behaviors. This is a challenge in feature extraction for Neural Network-based malware classifier. Because the key feature in an inline assembly will be lost when only API calls are extracted, then the classifier accuracy will be decreased.

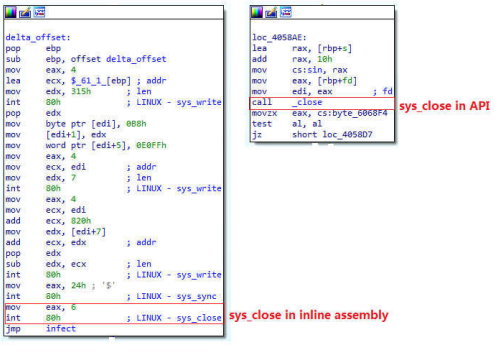

Fig. 1. Two forms of sys_close call

B. *Interpretability of Neural Network-based Methods*

After we mentioned the challenge of feature extracted in section II-A, we will describe the challenge of interpretability of Neural Network-based malware classifier. There are several explainable algorithms for Neural Network-based model, e.g., integrated gradients [19], DeepLIFT [20], SHAP [21] and Layer-wise Relevance Propagation (LRP) [23]. However, these explainable algorithms mainly focused on the computer vision domain. In the malware detection domain, the lack of transparency has introduced many black-box attacks. Malware developers can develop adversarial examples to evade these Neural Network-based malware detection systems. The challenge is to explain why the malware classifier labels the input sample malicious or benign, especially when the input sample is misclassified.

III. PROPOSED APPROACH

To mitigate these challenges mentioned in Section II, in this section, we mainly introduce the detail of our proposed malware detection system, which is shown in Fig. 2. Firstly, we describe the design principle of our proposed system. Secondly, show the components in detail.

As shown in Fig. 2, The architecture of our proposed malware detection system includes two phases: the training phase and the prediction phase. In the training phase, we train and output a model for the prediction phase. In the prediction phase, we use the trained model to predict whether the sample under test is malicious and benign, then explain the prediction. The important components are Extract Features Subsystem (in step 1), Classifier (in step 2), Interpreter (in step 4), as we will describe below.

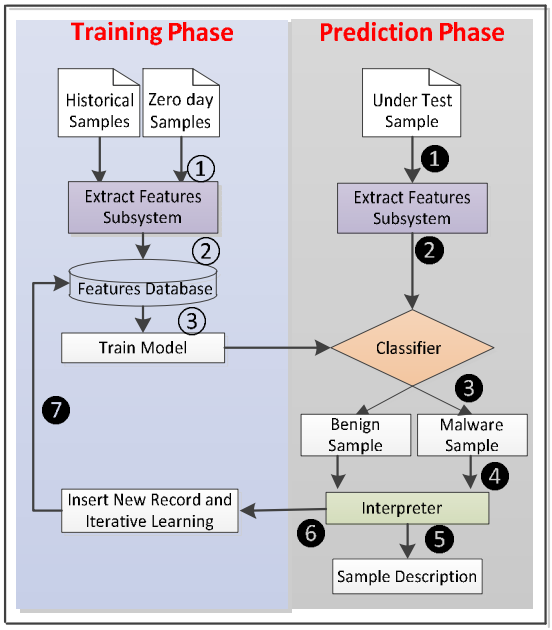

Fig. 2. The architecture of our proposed malware detection system

A. *Extract Features Subsystem*

We collect various historical malware from VirusShare [30] and VXHeavens [31]. To keep track of new emerging attacks, we dynamically add zero-day malware samples into the samples database every day. Benign executable samples can be collected from an open-source database (e.g., GitHub). We run malicious or benign samples on the same clean snapshot. The automatic features extract subsystem is shown in Fig. 3.

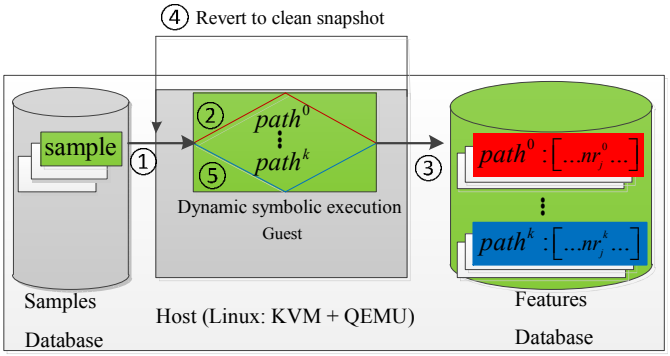

Fig. 3. Automatic features extract subsystem

We use a system-wide emulator QEMU [32] to emulate a standard computing node environment (e.g., server, personal computer, IoT devices, etc.) on a host machine of the Linux operating system. One clean guest snapshot is only for one sample. To traverse execution paths as many as possible, we use angr [33] to solve various input of a sample. As shown in Fig. 3, the red line and the yellow line represent different execution paths in a sample. We can accelerate to solve sample input in a parallel manner in an offline environment. We use an instrumentation tool to dynamic instrumentation the target sample. The instrumentation tool in different architecture is shown in Table I.

TABLE I. DIFFERENT ARCHITECTURE

Architecture	Instrumentation tool	System call table
ARM/AArch64	DynamoRIO [42]	arch/arm/tools/syscall.tbl [44]
MIPS32/ MIPS64	VG-MIPS [43]	arch/mips/kernel/syscalls/syscall_n64.tbl [45]
IA32/IA64	Intel Pintool [29]	arch/x86/entry/syscalls/syscall_64.tbl [34]
PowerPC32/ PowerPC64	Valgrind [50]	arch/powerpc/kernel/syscalls/syscall.tbl [51]

Our approach is architecture-agnostic because every platform has its instrumentation tool. We use an instrumentation tool to dynamic instrument the target sample, then the tool output system call number [34] trace of the target sample. The extracted features of the traditional system call-based malware detection approach [13], [16] is system call API, but the extracted features of our proposed approach are system call numbers. There are at least two advantages.

(1) The first advantage is that system call number trace can capture malicious behavior of malware which programmed with inline assembly, but API call or system call in the C standard library cannot. Moreover, Our approach does not need to care about API call differences in samples compiled with different version C standard libraries.

(2) The second advantage is that the system call number is an integer, which is easier and faster to process

than the system call API string. On the one hand, we could get a system call number in a hexadecimal form from the CPU without encoding/decoding API strings. It can speed up our malware detection algorithm. On the other hand, storing integers can reduce the features database size than API string, and the semantics information of system call number sequence is still preserved.

The system call number (NR) sequence of the k_{th} execution path ($path^k$) could be presented as:

$$NR^k = \left[nr_0^k, nr_1^k, ..., nr_j^k, ... \right], k \in N^+, j \in N^+ \quad (1)$$

Where nr_j^k is an integer from system call table in Linux source code which is shown in TABLE I, e.g., On IA64 platform [34], $nr_j^k \in \{0,1,...,547\}$, $nr_j^k = 0$ represents that the invoked system call is "sys_read" in k_{th} execution path at j_{th} moment.

B. Classifier

A d-dimensional feature vector $X^k \in \mathbb{R}^d$ represented the execution path ($path^k$) system call number sequence of a sample. A classifier f_w can be learned from the feature database. We use $f_w(X^k) = \hat{y}^k$ to denote the prediction process. Because the system call number (NR) sequence is a time series, we choose the Recurrent Neural Network (RNN) model to make a decision on whether a target sample is benign or malicious. Considering that the dependency between code in a binary file does not go only in one direction, we apply Bidirectional RNN (Bi-RNN) (e.g., Bi-SimpleRNN, Bi-GRU [35], Bi-LSTM [36]). The transform of the input feature is shown in Fig. 4.

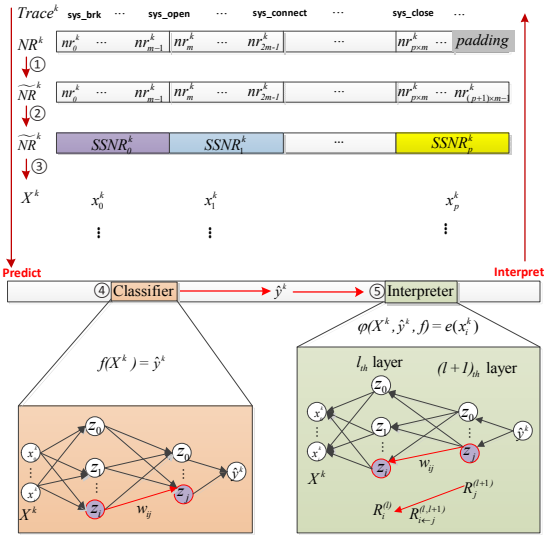

Fig. 4. Feature transform

The length of dynamic extracted system call number sequences (NR^k defined in Equation 1) usually varies a lot among different execution paths. To feed NR^k into the Bi-RNN model, we first pad each sequence NR^k to a fixed length $(p+1)\times m$. To reduce the size of the Bi-RNN model, we split each sequence into subsequences. Then NR^k (defined in Equation 1) can be presented as Equation 2:

$$\widetilde{NR}^k = [SSNR_0^k, SSNR_1^k, ..., SSNR_i^k, ..., SSNR_p^k] \quad (2)$$

$$SSNR_p^k = [nr_{p\times m}^k, nr_{p\times m+1}^k, ..., nr_{(p+1)\times m-1}^k] \quad (3)$$

Where $\widetilde{NR}^k$ is the system call number sequences after padding, $SSNR_p^k$ is the last subsequences of $\widetilde{NR}^k$, $p \in N^+$ is the total number of subsequences, $m \in N^+$ is the fixed length of each subsequence.

Because most samples have the same pattern of system call routine (e.g., the system call sequence of reading a file usually includes: sys_open, sys_read, sys_close), we can map $SSNR_i^k$ to a preset integer x_i^k.

$$x_i^k = g(SSNR_i^k), i \in N^+ \quad (4)$$

$$\widetilde{NR}^k = [x_0^k, x_1^k, ..., x_i^k, ..., x_p^k] \quad (5)$$

Where g is a map function, we can get the map by traversing all $SSNR_i^k$ in the features database. The key of the map is $SSNR_i^k$, and the value of the map is a unique identifier that is used to uniquely label the corresponding $SSNR_i^k$. The size of the map is equal to the total number of unique values in $SSNR_i^k, k \in N^+, i \in N^+$. Compare Equation 2 with Equation 5, we find that the input dimension of the training model is reduced from $K \times (p+1) \times m$ to $K \times (p+1)$, where K is the total number of samples fed into the training model.

C. Interpreter

As is commonly known, malware developers usually hide malicious code snippets in benign code to evade malware detection. It is often not easy for security analysts to find out x_i^k which makes the greatest contribution to the decision. We use $\varphi(X^k, \hat{y}^k, f_w) = e(x_i^k)$ to denote the explain process. The interpreter φ will output its explanation result $e(x_i^k)$, which represents the importance of the input feature x_i^k.

In the image classification area, Bach et al. [23] decomposed the non-linear classifier into several layers, then understood classification decisions by Layer-wise Relevance Propagation (LRP) algorithm on pixel-wise explanations. The

algorithm is shown in Fig. 5. The LRP algorithm is performed as Equation 6 and Equation 7 [37]. After the algorithm completes its execution, we can get an input feature that holds maximum relevance among all input features X^k and the model output $\hat{y}^k$.

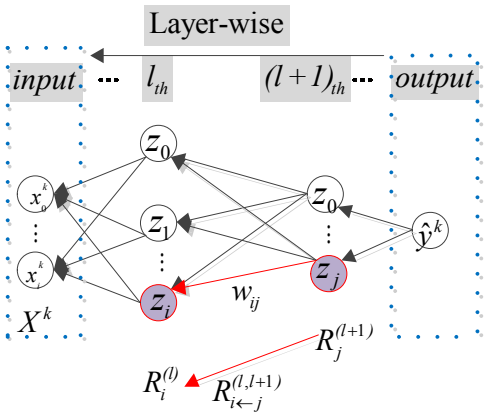

Fig. 5. Layer-wise Relevance Propagation (LRP) algorithm [23]

$$R_{i\leftarrow j}^{(l,l+1)} = \frac{z_i \cdot w_{ij} + \frac{\varepsilon \cdot sign(z_j) + \delta \cdot b_j}{N}}{z_j + \varepsilon \cdot sign(z_j)} \cdot R_j^{(l+1)} \quad (6)$$

$$R_i^{(l)} = \sum_j R_{i\leftarrow j}^{(l,l+1)} \quad (7)$$

In this paper, we adapt the LRP algorithm to help understand the predictions of Bi-LSTM. That is to say, our approach feed $\widetilde{NR}^k$ into the Bi-LSTM model, and output $\hat{y}^k$ which labeled the sample as benign or malicious, as shown in Fig. 4. With the LRP algorithm, we could get the relevance between x_i^k and $\hat{y}^k$. The x_i^k which hold the maximum relevance is usually the most important features. The corresponding $SSNR_i^k$ usually represents the malicious system call number. Then security analysts could catch these system call numbers when debugging the sample in the forensics phase. Our proposed algorithm could save analysts time and help analysts concentrate on these key system call numbers in reverse engineering.

IV. EXPERIMENTAL SETUP

To verify the effectiveness of our proposed approach for malware detection and explanation, we design and conduct extensive experiments to thoroughly evaluate our approach in this section. Firstly, we introduce our evaluation dataset. Then, we conduct experiments and compare evaluation metrics with baseline.

A. DataSet

Similar to previous works [53], [54], the experiment samples are two kinds of data sets: malicious samples and benign samples. Malicious samples were picked up from the VirusShare [30], which is a popular dataset used for malware analysis. We use VirusTotal [46] to identify the family of each malicious sample. For the benign samples, similar to previous works [17], [55], we use a variety of programs in the OS, i.e., system programs and user applications. Overall, there are 1523 malicious samples and 1427 benign samples. Both the malware samples and the benign samples are randomly divided into three subsets: training (60%), validation (20%), and testing (20%), as shown in TABLE II.

TABLE II. BENIGN AND MALICIOUS DATASET

Family		# Train	# Val	# Test	# Total	# Percent
Benign samples		857	285	285	1427	48.37%
Malicious samples	Virus	137	46	46	229	7.76%
	Backdoor	61	20	20	101	3.42%
	Trojan	103	33	33	169	5.73%
	Worm	88	29	29	146	4.95%
	other	528	175	175	878	29.77%
Overall		1774	588	588	2950	100%

B. Experimental Setup

Because other platforms (e.g., ARM, MIPS, PowerPC) have the same evaluation experiment design, we take the IA64 platform for malware detection in our evaluation. In the automatic features extract subsystem, the OS of the host machine is Ubuntu 16.04.7, the OS of the guest machine is Ubuntu 18.04.1. When the guest machine launch, we apply Intel PIN tools 3.7 [29] to dynamic instrumentation the target sample, then run the target sample in 10 minutes [40], because the majority of malware trigger their malicious behavior in their first several minutes. For example, the Slammer worm [41] infected more than 90% of vulnerable Internet hosts within 10 minutes. Then Pintools store system call number sequence trace on the guest file system. We share the trace file between the host and the guest with the 9P network protocol. The host machine adds these traces to the features database. Then we revert the guest machine to a clean snapshot for the next sample.

After the features database collected features of all historical samples, we feed these features to train a classifier model. The classifier includes traditional model (e.g. SVM, Random Forest) and deep learning model (e.g. Bi-SimpleRNN, Bi-GRU [35], Bi-LSTM [36]). We do not evaluate our approach on complex sequence-to-sequence models like Transformer [52] or BERT, because we argue simple models will be more secure and effective as the Unix Philosophy mentioned [48]. Evaluation experiments of all models are conducted in the same environment: a 64-bit Linux system with Linux kernel 4.4.0 running on an Intel i7-7800X quad-core processor (3.50 GHz) with 16 GB RAM. We trained all the models in this work on 1 Nvidia GTX 1080Ti GPU using the Keras package 2.2.4 [38] and with Tensorflow 2.1.0 [39] as backend, amounting to about 3,229 lines of Python code and 526 lines of C code.

C. Evaluation Metrics

Following the previous work [10], evaluation metrics could be shown as a confusing matrix in TABLE III, which labels an instance a positive class or a negative class.

TABLE III. CONFUSION MATRIX

Actual Class	Predicted Class	
	Positive	Negative
Positive	True Positive (TP)	False Negative (FN)
Negative	False Positive (FP)	True Negative (TN)

In TABLE III, the positive class represents a malicious sample, and the negative class represents a benign sample. True means the predicted class and the actual class are the same, and false means that the predicted class does not match the actual class [10]. Accordingly, we use F1-score as the evaluation metric:

$$F1 = 2\left(\frac{1}{Precision} + \frac{1}{Recall}\right)^{-1} with \begin{cases} Precision = TP/(TP + FP) \\ Recall = TP/(TP + FN) \end{cases} \quad (9)$$

V. EXPERIMENTAL RESULTS

In this section, firstly, we show the distribution of features that feed into our model. Secondly, we describe the performance comparison of different models and analyze the root cause. Thirdly, we show a case study with the interpretability of our model.

A. Distribution of Features

We collect a large number of system call number traces generated by benign or malicious samples in 10 minutes in the automatic features extract subsystem, which is shown in Fig. 3. We compare the length of the system call number trace. As shown in Fig. 6, the average length of the system call number trace generated by benign samples is 1103, and the average length of system call number trace generated by malware samples is 60299. That is to say, the malware makes more system calls than benign samples because the majority of malware triggers malicious behavior in their first several minutes to evade antivirus engines scanning periodically.

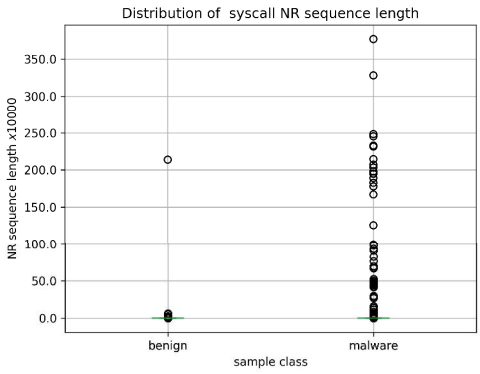

Fig. 6. Distribution of syscall NR sequence length

The first system call number (NR) made by malware samples under test is shown in Table IV. The first column is the first system call made by malware. The second column is the corresponding system call name. The third column is the corresponding malware sample hash from VirusShare [30] in our evaluation. We found that the first system call number (NR) of all benign samples is always 12 because 12 means that sys_brk helps program dynamically allocate memory in the heap. And the first system call number (NR) of malicious samples is various. The distribution of the first syscall NR is shown in Fig. 7. The reason is that benign sample developers follow a standard development process. While malicious sample developers often adopt unconventional coding techniques to evade detection systems. For example, the first system call NR of VirusShare_056e32863c0483922464fee50 b2bbd37 [30] is 1 (sys_write). Actually, the malware prints irrelevant strings on the terminal to cheat the user once it was launch.

TABLE IV. THE FIRST SYSCALL NR OF MALWARE

Syscall NR	Syscall Name	Malware sample hash [30] under test
1	write	056e32863c0483922464fee50b2bbd37
2	open	9d8ab409b27849f5a51aad1632c26e45
4	stat	7539917767b08c5949a61aedcd04d6b1
5	fstat	9fab551433152bd4eb67aa4ddce0c346
9	mmap	fcbfb234b912c84e052a4a393c516c78
12	brk	07a2c202f16f4fe2c92e768f460a1ddf
14	rt_sigprocmask	90c72abf054445578762fe412575df35
16	ioctl	942f7a3b5587415045ab5e25cc047466
20	writev	942f7a3b5587415045ab5e25cc047466
41	socket	a53203e408d2eb998a5f5a7eb25a1213
45	recvfrom	3c642c67b13745cac898eb7ab62f4b51
55	getsockopt	1f58ce5d091c450311458aa72a59be4a
63	uname	400d0b70661cab491b7241a33a962e8c
87	unlink	7cef5cec89742c08edc5439340dcd382
90	chmod	08d275864068a2e4291275bb6aeb10e9
122	setfsuid	c505ffa67c69dd80b58e17bf2e7b7dbc
125	capget	ab62ea3634bb269f2a734932cb5daf9b
136	ustat	bc05d911f80e7aeac52b3d908b943adb
157	prctl	3decf1b4e5e821c159e051a04fbf0452
158	arch_prctl	93f7192fe2961935eaf6568c655bc99f
221	fadvise64	f88f272c996f094cdd7f543962d6be15

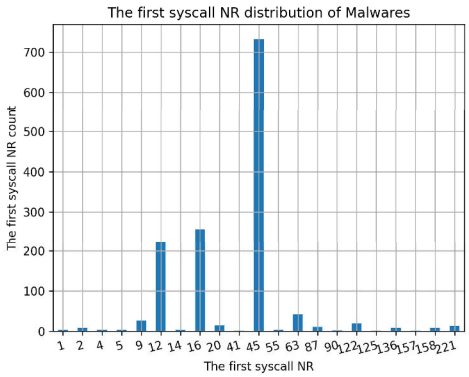

Fig. 7. Distribution of the first syscall NR

B. Performance Comparison

We implement 5 classification models, namely SVM, Random Forest, Bi-SimpleRNN, Bi-GRU [35], Bi-LSTM [36]. We describe the parameter of our implemented models in detail. In our implemented Support Vector Machine (SVM) model, the kernel function is the Radial Basis Function (RBF) kernel. The decision function is the original one-vs-one ('ovo') decision function. In our implemented Random Forest model, the number of trees in the forest is 10.

In the previous Bi-LSTM model, there are four layers, namely the input layer, the embedding layer, the bidirectional

LSTM, and the dense layer. In this paper, we feed two kinds of inputs into the same model, one kind is the original system call number sequence NR^k, and the other kind is the mapped system call number sequence $\widetilde{NR}^k$. We note the later model as Mapped Bi-LSTM (M-Bi-LSTM). The layer parameter of the two models is shown in Table V. The batch size is 32, the learning rate is 0.001, the optimizer is RMSprop [38], and the loss function is binary cross-entropy [38]. The Embedding layer size of M-Bi-LSTM is reduced by 20 times than vanilla Bi-LSTM. This demonstrates that our approach reduces the size of the neural network and speeds up the malware detection process. The unique difference between our M-Bi-SimpleRNN model and the traditional Bi-SimpleRNN is also the hidden layer, and the unique difference between our M-Bi-GRU model and our traditional Bi-LSTM is also the hidden layer.

In the Bi-LSTM model, we set the input size of the embedding layer is $BatchSize \times 1000$ because the average length of system call number sequence made by benign samples is 1103, and the average length of system call number sequence made by malware samples is 60299. Because the system call number is an integer from 0 to 547, which is defined in Linux source code [34], the vocabulary size of the embedding layer is set to 550.

TABLE V. BI-LSTM PARAMETER

Model	Layer	Input size	Layer parameter
Bi-LSTM	Embedding layer	BatchSize × 1000	Vocab size: 550 Embedding size: 16
	Bi-LSTM layer	BatchSize × 1000 × 16	Hidden layer size: 32
	Dense layer	BatchSize × 64	Size: 64 × 1 Activation: sigmoid
M-Bi-LSTM (Ours)	Embedding layer	BatchSize × 50	Vocab size: 9025 Embedding size: 16
	Bi- LSTM layer	BatchSize × 50 × 16	Hidden layer size: 32
	Dense layer	BatchSize × 64	Size: 64 × 1 Activation: sigmoid

In the M-Bi-LSTM model, the vocabulary size of the embedding layer is 9025 because the size of the map is 9025. That is to say, there are 9025 different system call number sequence slices in our feature database. The vocabulary size of the embedding layer in the M-Bi-LSTM model is larger than the Bi-LSTM model because of the map in Equation 4. But the input size of the embedding layer is reduced by m times, which is defined in Equation 3, in this evaluation m is 20. Then the time cost by the classifier is reduced. In our evaluation experiment, both malware samples and benign samples are randomly divided into three subsets: training (60%), validation (20%), and testing (20%). The total number of the training sample is 1774, the total number of the validation sample is 588, the total number of the test sample is 588.

We conduct our experiment 10 times to get the average evaluation metrics. The average F1-score of models in our evaluation is shown in Fig. 8. We note the Bi-LSTM model obtains the highest accuracy (98.45%), and the accuracy of the M-Bi-LSTM model decreases a little (97.39%). The reason behind this is that our approach label m system call number $nr^k_{i\times j}, j \in [m, 2m-1]$ in $SSNR^k_i$ as the same time tag.

We note the M-Bi-LSTM in Fig. 8 obtains 97.35% accuracy and takes nearly 0.95 milliseconds (in Fig. 9) to predict a sample. The average time of our M-Bi-LSTM model is reduced by nearly 4.7 times than the Bi-LSTM model. Random Forest takes less time than Bi-LSTM. The reason is that Random Forest is an ensemble technique that trains a group of decision trees with parallel computing. But the explanation of Random Forest classifier output is also random. We note the M-Bi-LSTM (in Fig. 10) obtains the highest AUC value (0.9731). We argue that the M-Bi-LSTM model is more useful than the other nine models in malware detection.

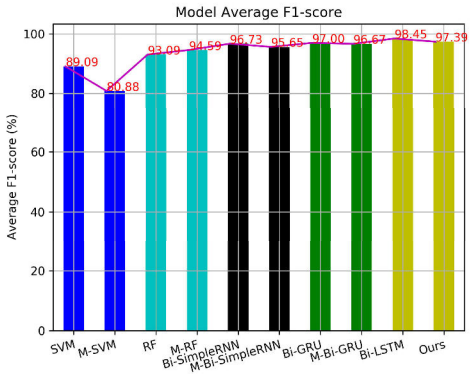

Fig. 8. The average F1-score of our experiment models

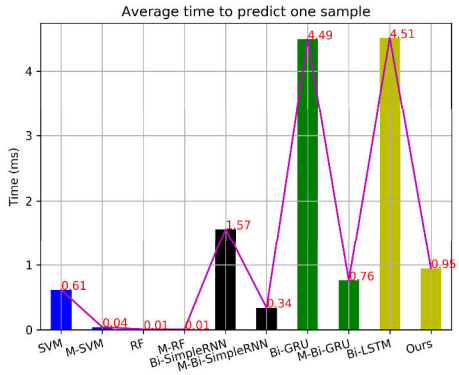

Fig. 9. The average time of our experiment models

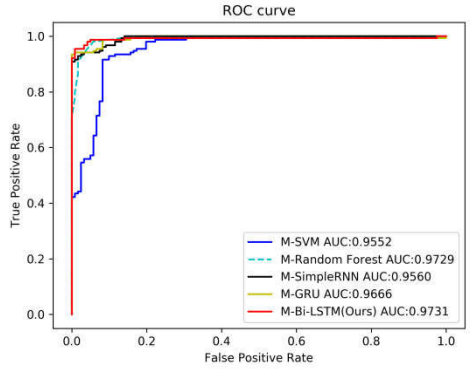

Fig. 10. The ROC curve of our experiment models

Feature	x0	x1	x2	x3	x4	x5	x6	x7	···										
Importance	49	50	3	2	1	6	4	5	···										
Mapped NR	5053	2956	6805	5649	6316	3402	1829	1337	1413	8754	8468	1005	1250	4245	6693	2833	8111	3297	1967
	8832	9023	9023	9023	9023	9023	9023	9023	9023	9023	9023	9023	9023	9023	9023	9023	9023	9023	9023
	9023	9023	9023	9023	9023	9023	9023	9023	9023	9023	9023	9023							

Fig. 11. The explanation of malicious sample prediction

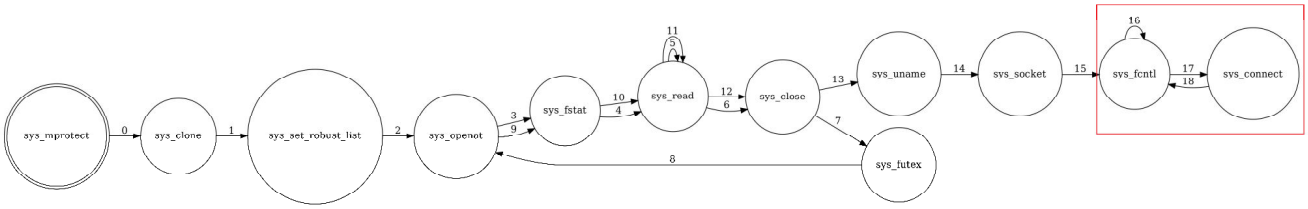

Fig. 12. The call graph of system call sequence slices

C. *Interpretability Analysis*

After the classifier output a prediction, we apply the Layer-wise Relevance Propagation (LRP) algorithm [23] in our M-Bi-LSTM model to find which system call number sequence slice (x_i^k defined in Equation 4) makes the greatest contribution in the malware decision.

As the detailed description of the explanation method with a case study in LEMNA [26], we will show our explanation method in detail with a malware case. We choose VirusShare_4cafaf20c27c828974f9abd32d921f70 [30] which is a malware sample of Trojan [30] family. As is shown in Fig. 11, the mapped system call number sequence made by the sample is [5053, 2956, 6805, 5649,…,9023], which the integer 9023 is the padding system call number to feed our M-Bi-LSTM model. Then the model output a prediction that the sample is malicious. We apply the LRP algorithm [23] to explain the sample prediction, which is shown in Fig. 11. The mapped NR in red color has a positive influence on the malware decision process, and the mapped NR in blue color has a negative influence. In Fig. 11, we note x_4 makes the greatest contribution in malware decision, then we look up the Mapping Table (defined in Equation 4). The corresponding system call name sequence of x_4 in the mapping table is ['sys_mprotect', 'sys_clone', 'sys_set_robust_list', 'sys_openat', 'sys_fstat', 'sys_read', 'sys_read', 'sys_close', 'sys_futex', 'sys_openat', 'sys_fstat', 'sys_read', 'sys_read', 'sys_close', 'sys_uname', 'sys_socket', 'sys_fcntl', 'sys_fcntl', 'sys_connect', 'sys_fcntl']. The call graph of the system call sequence in x_4 is shown in Fig. 12.

We use strace utility [49] and GDB utility to verify the interpretability of our approach. The system call trace collected by strace utility [49] is shown in Fig. 13. The system call sequences slices corresponding to x_4 is from line 82 to line 101, we can find that the sample is trying to read the DNS setting file "/etc/resolv.conf" on line 91 and send the file to a remote server on line 100 in Fig. 13. We decompiled the sample with the IDA Pro. The part decompiled code of the malicious sample is shown in Fig. 14. We can see the key malicious code is line 37 in the red box of Fig. 14. The red box in Fig. 12 shows the same malicious system call sequence slices. This demonstrated our proposed approach successfully captures key malicious code snippets.

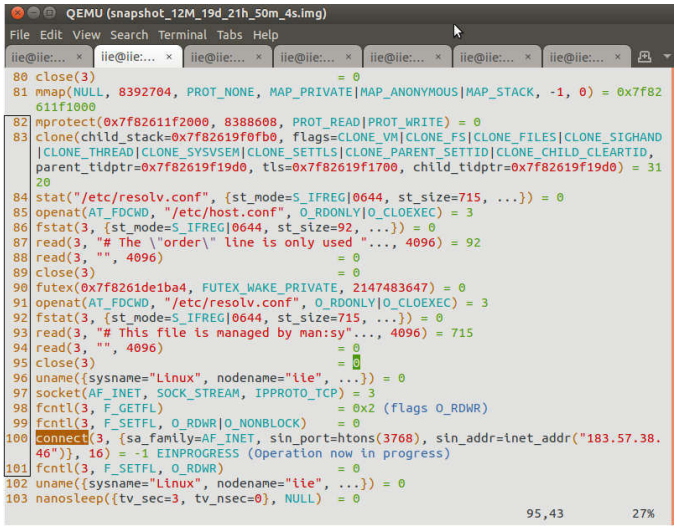

Fig. 13. The system call trace of VirusShare_4caf

```
26    memset(&s, 0, 0x28uLL);
27    strncpy(&s, v8->if_name, 0xFuLL);
28    if ( ioctl(fd, 0x8915uLL, &s) >= 0 )
29    {
30      *(_QWORD *)sin = &v5;
31      close(fd);
32      if ( byte_6068F4 )
33        CreateAutoRun();
34      pthread_create(&newthread, 0LL, F_Main, 0LL);
35      while ( 1 )
36      {
37        _ConnectServer();
38        usleep(0x3E8u);
39      }
40    }
```

Fig. 14. The part decompiled code of VirusShare_4caf

With the help of our approach, security analysts can set a breakpoint before key system call when debugging with GDB. Debugging the malicious sample is shown in Fig. 15. We can catch the "connect" system call, which is from our interpreter

output. This demonstrated our proposed approach is effective for saving security analysts time to locate the key malicious code snippets.

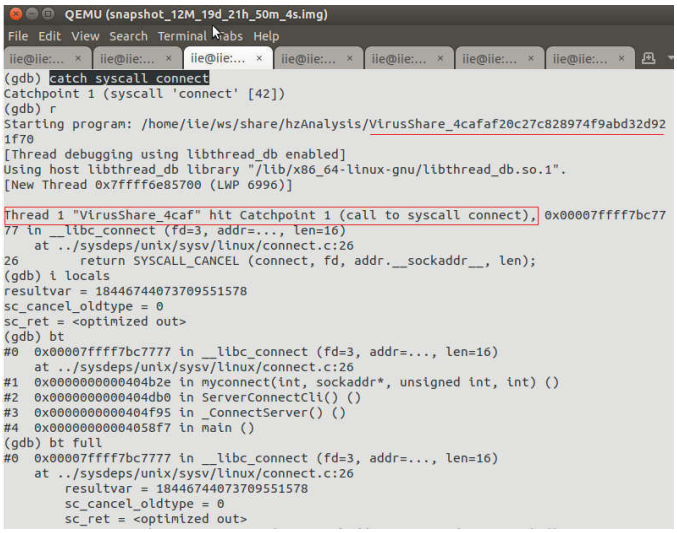

Fig. 15. The GDB debug of VirusShare_4caf

VI. CONCLUSIONS

In this paper, we propose an effective approach for malware detection and explanation, which can locate malicious code snippets by explaining the malware classifier decision result. Our approach can reduce the input dimension of the training model by mapping input time series and reduce the time of model prediction. We conduct extensive experiments on malware classification tasks with our proposed M-Bi-LSTM model. Then we adopt the LRP algorithm to save the malware analyst time to locate malicious code snippets. The experimental results demonstrate that our proposed approach performs better than previous methods.

REFERENCES

[1] B. Yuan et al., "Shattered Chain of Trust: Understanding Security Risks in Cross-Cloud IoT Access Delegation," in 29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020, 2020, pp. 1183–1200.

[2] A. Costin, J. Zaddach, A. Francillon, and D. Balzarotti, "A Large-Scale Analysis of the Security of Embedded Firmwares," in Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014, 2014, pp. 95–110.

[3] "AV-TEST—The Independent IT-Security Institute."https://www.av-test.org/en/statistics/malware/, Accessed:2020.

[4] D. Korczynski and H. Yin, "Capturing Malware Propagations with Code Injections and Code-Reuse Attacks," in Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017, 2017, pp. 1691–1708.

[5] A. Bulazel and B. Yener, "A Survey On Automated Dynamic Malware Analysis Evasion and Counter-Evasion: PC, Mobile, and Web," in *Proceedings of the 1st Reversing and Offensive-oriented Trends Symposium on - ROOTS*, Vienna, Austria, pp. 1–21, 2017.

[6] K. Ying, T. Luo, Z. Su, and X. Xing, "Superman Powered by Kryptonite: Turn the Adversarial Attack into Your Defense Weapon," In *Blackhat*, 2020, p. 11.

[7] H. S. Anderson, A. Kharkar, B. Filar, D. Evans, and P. Roth, "Learning to Evade Static PE Machine Learning Malware Models via Reinforcement Learning," CoRR, vol. abs/1801.08917, 2018, [Online]. Available: http://arxiv.org/abs/1801.08917.

[8] H. Alasmary et al., "Soteria: Detecting Adversarial Examples in Control Flow Graph-based Malware Classifiers," In ICDCS, 2020, p. 11.

[9] E. Amer and I. Zelinka, "A dynamic Windows malware detection and prediction method based on contextual understanding of API call sequence," In *Computers & Security*, vol. 92, pp. 101760, 2020.

[10] Xiaofeng, L., Fangshuo, J., Xiao, Z., Shengwei, Y., Jing, S., Lio, P., "ASSCA: API sequence and statistics features combined architecture for malware detection." *Comput. Netw.* vol.157, pp.99–111, 2019.

[11] A. K. Marnerides, P. Spachos, P. Chatzimisios, and A. Mauthe, "Malware detection in the cloud under ensemble empirical model decomposition," in *Proc. 6th IEEE Int. Conf. Netw. Comput.*, pp. 82–88, 2015.

[12] W. Han, J. Xue, Y. Wang, L. Huang, Z. Kong, and L. Mao, "MalDAE: Detecting and explaining malware based on correlation and fusion of static and dynamic characteristics," Comput. Secur., vol. 83, pp. 208–233, 2019, doi: 10.1016/j.cose.2019.02.007.

[13] D. Kirat and G. Vigna, "MalGene: Automatic Extraction of Malware Analysis Evasion Signature," in Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015, 2015, pp. 769–780.

[14] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural Network-based Graph Embedding for Cross-Platform Binary Code Similarity Detection," in Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017, 2017, pp. 363–376.

[15] R. Paleari, L. Martignoni, G. F. Roglia, and D. Bruschi, "A fistful of red-pills: How to automatically generate procedures to detect CPU emulators," In WOOT'09 Proceedings of the 3rd USENIX Workshop on Offensive Technologies. 2009, p. 7.

[16] D. Kirat, G. Vigna, and C. Kruegel, "BareCloud: Bare-metal Analysis-based Evasive Malware Detection," in Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014, 2014, pp. 287–301.

[17] M. Ozsoy, C. Donovick, I. Gorelik, N. B. Abu-Ghazaleh, and D. V. Ponomarev, "Malware-aware processors: A framework for efficient online malware detection," in 21st IEEE International Symposium on High Performance Computer Architecture, HPCA 2015, Burlingame, CA, USA, February 7-11, 2015, 2015, pp. 651–661.

[18] N. Sehatbakhsh *et al.*, "REMOTE: Robust External Malware Detection Framework by Using Electromagnetic Signals," *IEEE Transactions on Computers*, vol. 69, no. 3, pp. 312–326, 2020.

[19] M. Sundararajan, A. Taly, and Q. Yan, "Axiomatic attribution for deep networks," in ICML, 2017, pp. 3319–3328.

[20] A. Shrikumar, P. Greenside, and A. Kundaje, "Learning important features through propagating activation differences," in ICML, 2017, pp. 3145–3153.

[21] S. M. Lundberg and S.-I. Lee, "A unified approach to interpreting model predictions," in NIPS, 2017, pp. 4765–4774.

[22] M. D. Zeiler and R. Fergus. "Visualizing and understanding convolutional networks," In *ECCV* 2014, pp: 818–833, 2014.

[23] S. Bach, A. Binder, G. Montavon, F. Klauschen, K.-R. Müller, and W. Samek, "On Pixel-Wise Explanations for Non-Linear Classifier Decisions by Layer-Wise Relevance Propagation," PLoS ONE, vol. 10, no. 7, p. e0130140, Jul. 2015.

[24] M. T. Ribeiro, S. Singh, and C. Guestrin, "'Why Should I Trust You?': Explaining the Predictions of Any Classifier," in Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016, 2016, pp. 1135–1144.

[25] M. Sundararajan, A. Taly, and Q. Yan, "Axiomatic Attribution for Deep Networks," in Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017, 2017, vol. 70, pp. 3319–3328.

[26] W. Guo, D. Mu, J. Xu, P. Su, G. Wang, and X. Xing, "LEMNA: Explaining Deep Learning based Security Applications," in Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018, 2018, pp. 364–379.

[27] Z. Xu, S. Ray, P. Subramanyan, and S. Malik, "Malware detection using machine learning based analysis of virtual memory access patterns," in Design, Automation & Test in Europe Conference & Exhibition, DATE 2017, Lausanne, Switzerland, March 27-31, 2017, 2017, pp. 169–174.

[28] H. Sayadi, N. Patel, S. M. P. D, A. Sasan, S. Rafatirad, and H. Homayoun, "Ensemble learning for effective run-time hardware-based malware detection: a comprehensive analysis and classification," in Proceedings of the 55th Annual Design Automation Conference, DAC 2018, San Francisco, CA, USA, June 24-29, 2018, 2018, p. 1-6.

[29] Intel, "Pin 3.7 User Guide." https://software.intel.com/sites/landingpage/pintool/docs/97619/Pin/html/, Accessed:2020.

[30] "VirusShare." https://virusshare.com/. Accessed: 2020.

[31] "Vxheavens." http://ww1.vxheavens.com/. Accessed: 2020.

[32] F. Bellard, "QEMU, a Fast and Portable Dynamic Translator," In *Proceedings of the annual conference on USENIX Annual Technical Conference*, 2005.

[33] "angr," https://github.com/angr/angr, Accessed:2020.

[34] Linux, "Linux 64-bit system call numbers and entry vectors," https://github.com/torvalds/linux/blob/master/arch/x86/entry/syscalls/syscall_64.tbl, Accessed:2020.

[35] Cho Kyunghyun, Van Merrienboer Bart, Gulcehre Caglar, Bahdanau Dzmitry, Bougares Fethi,et al. "Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation," In *EMNLP*, pp, 1724—1734, 2014.

[36] Hochreiter, S., Schmidhuber, J., "Long short-term memory," *Neural Comput*. vol.9, No.8, pp. 1735-1780, 1997.

[37] Leila Arras, Gregoire Montavon, Klaus-Robert Müller, and Wojciech Samek, "Explaining Recurrent Neural Network Predictions in Sentiment Analysis," In Proceedings of the 2017 EMNLP Workshop on Computational Approaches to Subjectivity, Sentiment and Social Media Analysis (WASSA), pp. 159–168, 2017.

[38] CHOLLET, F., ET AL. "Keras," https://keras.io/, Accessed: 2020.

[39] ABADI, M., BARHAM, P., CHEN, J., CHEN, Z., DAVIS, A., DEAN, J., DEVIN, M., GHEMAWAT, S., IRVING, G., ISARD, M., ET AL. "Tensorflow: a system for large-scale machine learning". In *OSDI*, 2016.

[40] A. Costin, J. Zaddach, and S. Antipolis, "IoT Malware: Comprehensive Survey, Analysis Framework and Case Studies," In *Blackhat*, 2018.

[41] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver, "Inside the slammer worm," *IEEE Security & Privacy*, vol. 99, no. 4, pp. 33–39, 2003.

[42] "DynamoRIO," https://dynamorio.org/, Accessed: 2020.

[43] Z. Anwar, M. Sharf, E. Khan, and M. Mustafa, "VG-MIPS: A dynamic binary instrumentation framework for multi-core MIPS processors," in *INMIC*, pp. 166–171, 2013.

[44] Linux, "ARM system call table," https://github.com/torvalds/linux/blob/master/arch/arm/tools/syscall.tbl, Accessed: 2020.

[45] Linux, "MIPS system call table," https://github.com/torvalds/linux/blob/master/arch/mips/kernel/syscalls/syscall_n64.tbl, Accessed: 2020.

[46] "VirusTotal." https://virustotal.com/. Accessed: 2020.

[47] R. O'Neill, "Learning Linux binary analysis: uncover the secrets of Linux binary analysis with this handy guide," *Birmingham, Mumbai: Packt Publishing*, pp. 94–95, 2016.

[48] Raymond, Eric. "The Unix Philosophy in One Lesson". The Art of Unix Programming, *Addison-Wesley*, pp. 48-49, 2009.

[49] "strace(1)—Linux manual page", https://man7.org/linux/man-pages/man1/strace.1.html, Accessed:2020.

[50] Valgrind, "Valgrind Documentation," https://valgrind.org/, Accessed: 2020.

[51] Linux, "PowerPC system call table," https://github.com/torvalds/linux/blob/master/arch/powerpc/kernel/syscalls/syscall.tbl, Accessed: 2020.

[52] S. Garg, T. Vu, and A. Moschitti, "TANDA: Transfer and Adapt Pre-Trained Transformer Models for Answer Sentence Selection," In *AAAI*, vol. 34, no. 05, pp. 7780–7788, Apr. 2020.

[53] F. li, J. Han, Z. Zhu, and D. Meng, "Spatial-Temporal Attention Network for Malware Detection Using Micro-architecture Features," in *2019 International Joint Conference on Neural Networks (IJCNN)*, Budapest, Hungary, Jul. 2019, pp. 1–8.

[54] F. Li, C. Yan, Z. Zhu, and D. Meng, "A Deep Malware Detection Method Based on General-Purpose Register Features," in *Computational Science* – ICCS 2019, vol. 11538, pp. 221–235.

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