Computers & Security 122 (2022) 102871

Contents lists available at ScienceDirect

Computers & Security

journal homepage: www.elsevier.com/locate/cose

# Efficient and Robust Malware Detection Based on Control Flow Traces Using Deep Neural Networks

Weizhong Qiang[a,b,*], Lin Yang[a,b], Hai Jin[a,c]

[a] National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, Wuhan, 430074, China
[b] School of Cyber Science and Engineering Huazhong University of Science and Technology, Wuhan, 430074, China
[c] School of Computer Science and Technology Huazhong University of Science and Technology, Wuhan, 430074, China

A R T I C L E  I N F O

*Article history:*
Received 24 April 2022
Revised 10 July 2022
Accepted 6 August 2022
Available online 8 August 2022

*Keywords:*
Malware detection
Program packing
Deep learning
Intel processor trace

A B S T R A C T

Nowadays, the rapid growth of the number and variety of malware brings great security challenges. Machine learning has become a mainstream tool for effective malware detection, which can mainly be classified into static and dynamic analysis methods.

The purpose of malware detection is to have a good and stable detection performance for different software forms. However, many static analysis methods are easily affected by packing and other code obfuscation techniques, and dynamic analysis methods are commonly believed more robust, while the impact of packing on them has received little attention. In addition, adversarial sample attacks against dynamic analysis methods have also been conducted. This indicates the need to investigate more accurate and robust malware classification methods.

In this paper, we propose a new robust dynamic analysis method for malware detection by using specific fine-grained behavioral features, i.e., control flow traces. Further, a malware classifier is constructed by converting control flow traces into byte sequences and applying a combination of *convolutional neural networks* and *long short term memory*.

The proposed classifier can effectively detect malware with an accuracy of up to 95.7%, as well as detect unseen malware with an accuracy of 94.6% (indicating a good performance in handling the evolution of malware). Meanwhile, it is first found experimentally that packing has specific interference with existing behavior-based malware classifiers, resulting in even worse performance than static classifiers in some cases. However, the proposed classifier performs well in terms of robustness, showing stable performance under the interference of an uneven packing distribution in the dataset, with a 26.3% higher true positive rate for unpacked samples compared to the API call-based classifier. In addition, the classifier is also more robust against adversarial samples, with a detection rate of at least 83%.

## 1. Introduction

Malicious programs cause security breaches by stealing data or damaging systems. In McAfee's report (McAfee, 2021), in the first quarter of 2021, more than 87 million new malicious programs were discovered, and the number of new malicious signed binary files exceeded 930,000. The average number of suspicious files uploaded to VirusTotal is over one million per day (VirusTotal, 2020). As a result, malware detection has always been an important research topic. Security vendors have traditionally relied on signature-based methods to manually generate signature rules for malware detection, which usually fail to detect new malware or unsigned variants. The exponential growth in the number of malicious samples and the emergence of new variants (many of which are various obfuscated versions of known samples) make malware detection even more challenging. Therefore, the ultimate target of malware detection is to have higher detection performance in the presence of newer malware and more different forms of malware obfuscation.

Machine learning has been widely used to automatically detect malware at scale (Krčál et al., 2018; Raff et al., 2017, 2018; Saxe and Berlin, 2015). When designing a neural network-based malware classifier, the first step is to determine the feature set, which can be obtained through static or dynamic analysis. Static analysis methods are fast and efficient in detection, but they are

[*] Corresponding author.
*E-mail address:* wzqiang@hust.edu.cn (W. Qiang).

https://doi.org/10.1016/j.cose.2022.102871
0167-4048/© 2022 Elsevier Ltd. All rights reserved.

usually susceptible to code obfuscation. That is, attackers can effectively bypass static detection by using simple code-padding techniques (Fleshman, 2019; Kolosnjaji et al., 2018), of which packing obfuscation is more common in malware. A recent study (Aghakhani et al., 2020) shows that the deviation in the packing distribution of benign and malicious samples in the training set will cause static classifiers to learn specific packing routines as a sign of maliciousness.

Compared to static analysis methods, dynamic analysis methods focus on recording the execution behavior and are often considered to be more resistant to interference and robust. Because the behavior of a program can be reflected by its calls to the underlying system APIs, most previous work has traced the API calls and represented the result in the form of ordered sequences (Hu and Tan, 2017b; Huang and Stokes, 2016; Rosenberg et al., 2018) or directed graphs (Firdausi et al., 2010; Fredrikson et al., 2010). It is generally assumed in previous studies that dynamic analysis is more robust to code obfuscation, such as the popular packing techniques in malware.

However, whether and how much packing has an impact on dynamic analysis has typically been ignored by previous studies. We believe that this assumption is problematic because packing affects not only the static features of programs, but also the behavior of programs during execution. In addition to packing, the emergence of effective black-box attacks against neural networks also makes dynamic methods based on API call less robust (Hu and Tan, 2017a,b; Rosenberg et al., 2020, 2018). Specifically, by modifying the API call sequence of malware, an attacker can generate malware samples, which can lead to false detection results.

We argue that effectiveness and robustness are two important metrics of malware detectors, and existing research efforts have focused mainly on effectiveness and have not explored robustness sufficiently. Therefore, improving the robustness of malware detectors against packing and adversarial samples while ensuring their effectiveness is our key research question.

In this paper, we proposed a deep learning-based malware detection method that uses a new type of behavioral feature, namely low-level control flow traces of program execution, to identify malicious programs. Feature extraction is performed by tracing the direction of conditional branches, indirect branch targets and instruction pointers based on the hardware feature, Intel Processor Trace (IPT), and representing the control flow traces as sequences of behavioral bytes. Then, the model are constructed based on *Convolutional Neural Network* (CNN) and *Long Short Term Memory* (LSTM), where CNN can solve the memory usage problem and can capture higher-level information in local contexts, while LSTM can preserve the semantic relationships between local contexts so as to make a decision through the whole sequence.

In the evaluation, the proposed classifier can achieve good effectiveness in malware detection, even in the case of malware evolution. In terms of robustness, experiments were conducted to test the performance of the proposed classifier for the interference of packing on malware detection, and the vulnerability of detection to adversarial samples, respectively, and the proposed classifier showed better robustness.

In summary, the main contributions of this paper are as the following:

- We proposed a deep learning-based method combining CNN and LSTM to detect malware by using control flow traces during program execution. The proposed method has higher detection performance, including in the presence of newer malware.
- We found that the dynamic analysis method based on API calls is not as robust as commonly believed under obfuscations such as packing. In contrast, our classifier based on control flow traces is more robust to the interference of the uneven packed ratio.
- The proposed method is robust to state-of-the-art adversarial attacks. For adversarial samples that can successfully evade API call-based detection, our classifier can detect them effectively.

The structure of this study is described as follows: Section 2 introduces related work; Section 3 introduces the proposed method; the datasets is described in Section 4; Section 5 shows the evaluation; the last section gives the conclusion.

## 2. Related work

Deep learning helps reduce the onerous manual work that exists in traditional malware analysis methods and has been used to discover potential patterns in a large-scale instance to make malware detection more efficient (Kephart et al., 1995). Research in this field generally consists of two categories, static analysis methods and dynamic analysis methods. Without attempting to execute malware, static analysis methods aim at extracting features from binary programs and can be used to provide pre-execution detection and prevention. In contrast, for dynamic analysis methods, researchers focused on extracting features of interest from the execution of programs. The approach is usually to monitor malware in a virtual environment (such as a sandbox) to record activities and generate a behavior analysis report (Kolosnjaji et al., 2016; Jindal et al., 2019). Researchers extract useful features from reports to describe malicious behavior.

*Static analysis methods.* In static analysis, various features have been proposed to represent samples for classification. Among them, parsing *Portable Executable* (PE) binary files is a relatively common approach. Hardy et al. (2016) extracted each API function in the *Import Address Table* and converted it into a fixed global ID, which can represent each PE file as a fixed-length feature vector. Using more professional knowledge, Saxe and Berlin (2015) extracted various features from the PE files, including the *Import Address Table*, the metadata of PE header, etc. but their method does not perform well on unseen samples. When they divided the samples in the dataset according to the timestamp, and the test was performed on the samples after the timestamp, the true positive rate drops from 95.2% to 67.7%.

Another common method is to directly use the raw bytes of the binaries as features, which requires little knowledge and can be widely used in various file types. Nataraj et al. (2011a,b) introduced an image conversion method to visualize executable files, relying on the similarity between malware family images to build classifiers with CNN. In contrast, Le et al. (2018) converted the raw bytes of the binary file into one-dimensional representation without any need for feature extraction.

Raff et al. (2018) proposed the MalConv model. The model is trained on raw byte sequences of binaries and it is the first neural network model applied in the field of malware detection to process long input sequences.

A recent study (Aghakhani et al., 2020) has shown that packers have a great influence on deep learning based methods using static analysis features, and the imbalance of the distribution of packed samples will make the classifier bias towards packing detection rather than malicious detection. However, because automatic unpacking techniques usually lag behind the packing technology, and manual unpacking is difficult and inefficient, program unpacking is often full of challenges. Moreover, in order to protect copyright, packed benign programs have become more common (Aghakhani et al., 2020; Rahbarinia et al., 2017), causing the problems faced by static analysis methods to become more prominent. Using other obfuscation techniques, Kolosnjaji et al. (2018) successfully attacked the malware classifiers based on raw bytes. They

**Table 1**
Comparison of related work on malware detection.

| Compared work | Algorithms | Features | Type | Robustness to packing considered? |
|---|---|---|---|---|
| Hardy et al. (2016) | Stacked autoencoders | API from PE files | Static | No |
| Saxe and Berlin (2015) | Deep feedforward network | PE metadata | Static | No |
| Nataraj et al. (2011a,b) | K-nearest neighbors,CNN | Byte sequences to image | Static | No |
| Le et al. (2018) | CNN,CNN+LSTM | Raw byte sequences | Static | No |
| Raff et al. (2018) | MalConv | Raw byte sequences | Static | No |
| Firdausi et al. (2010) | K-nearest neighbors,SVM | API call sequence | Dynamic | No |
| Ronghua et al. (2010) | Random forest | API call sequence | Dynamic | No |
| David and Netanyahu (2015) | Deep belief network | API calls and other events | Dynamic | No |
| Tobiyama et al. (2016) | LSTM, CNN | API calls to image | Dynamic | No |
| Ficco (2022) | Multimodal DNN | API calls, memory, traffic | Dynamic | No |
| Our work | LSTM, CNN | Control flow traces | Dynamic | Yes |

generated adversarial samples by padding a few optimized bytes at the end of the malicious binaries, and attained an evasion rate of 60%. Similarly, Fleshman (2019) successfully made all 50 malicious PE files evade the detection of two static features based classifiers, namely Raff et al. (2018) and Anderson and Roth (2018), through code injection in a competition (Anderson, 2019).

*Dynamic analysis methods.*

Many kinds of features are used in dynamic analysis, among which the more common one is to use the API call sequence traced during program execution as the detection feature. Firdausi et al. (2010) conducted experiments on a dataset with a small number of samples based on API call sequences, and applied the decision tree algorithm for classification. Ronghua et al. (2010) used the same feature and obtained higher accuracy based on the random forest algorithm. David and Netanyahu (2015) executed malicious samples in the sandbox to trace API calls and other events, then applied *Deep Belief Network* (DBN) for classification. In a slightly different way, Tobiyama et al. (2016) used LSTM to convert a series of API calls extracted from behavior logs into feature images that contain local features representing process events and then used CNN to perform image classification.

However, recent studies show that API call-based analysis methods have been effectively attacked by adversarial samples. Rosenberg et al. (2018) created adversarial samples by adding useless APIs to malicious binaries, and successfully attacked various machine learning-based classifiers, with a maximum evasion rate of 98%. Hu and Tan (2017b) used *Generative Adversarial Network* (GAN) to generate adversarial API sequences to bypass the RNN-based detector. Therefore, it is also an important topic to propose a robust classifier to effectively deal with adversarial samples. The method of ensemble detection (Ficco, 2022), which uses several different detectors and switches between them randomly, allows more resilient detection against the evasion of adversarial samples. However, the effect of packing on detectors has not been studied in this type of methods.

The comparison of some of the related work is presented in Table 1, and to the best of our knowledge, most of the existing works have not investigated whether packing has an impact on the robustness of machine learning-based methods, or have not considered how to address the impact of packing on robustness. In particular, some works have not described the distribution of packed samples in the datasets, while others simply discard the packed samples.

In contrast, we first found that packing will affect not only the static properties of a program, but also its dynamic execution behavior. Moreover, we proposed a dynamic analysis method based on the feature of control flow traces that can provide significant robustness against packing and adversarial samples.

## 3. Proposed method

In this section, we present a dynamic analysis approach based on the program's control flow traces for malware detection. Firstly, *Intel Processor Trace* (IPT) is used to generate the execution trace packets of the program and the decoder is utilized to decode the packets. Secondly, the decoded data is transformed into input vectors for learning. We adopted a light-weight method to convert the decoded control flow data into one-dimensional behavioral byte sequences, and used an embedding layer to map byte sequences to embedding vectors.

Thirdly, the deep learning architecture is designed that can scale well to adapt to long feature sequences and provide excellent malware detection performance. Due to the feature of hardware execution tracing, our method is cross-system and has good portability. As the low-level control flow traces are fine-grained, how to attack the control flow is a difficult task for attackers.

### 3.1. Feature processing

IPT is a low-overhead hardware feature that can be used to record control flow transfers of program execution. Control flow information is collected in data packets, which mainly include the generated *Taken Not-Taken* (TNT) and *Target IP* (TIP) packets. TNT packets record the direction of direct jump instructions, and TIP packets record the target IP of indirect branches, exceptions, interrupts, and other branches or events. These data packets can be parsed in a fixed format by the decoder.

**TNT packet handling.** The length of a TNT packet is 8 bits, including a 1-bit header, 6-bit information flags (recording the jump of instructions) in the middle, 1-bit on the tail (representing stop flag of the TNT packet).

For each TNT packet, the payload is extracted and concatenated together. Then, the sequence is divided by 8 bits, with the last part less than 8 bits being padded with 0.

Finally, each 8-bit segment is converted into an unsigned positive integer (the value is [0, 255]) with a size of 1 byte.

**TIP packet handling.** In a TIP packet, a highly compressed method is used to store the target address of the indirect branch, by discarding the same prefix as the last saved IP, and then recording the remaining part. When parsing the TIP packet, the upper (most effective) address bytes of the previous IP is combined with the current payload to restore the complete target address.

Due to the widespread use of *Address Space Layout Randomization* (ASLR), when tracing the program execution, the base address of the image is recorded. For each reconstructed IP, its offset from the base address is calculated, and then each offset is converted into unsigned integers in the same way as TNT packets are pro-

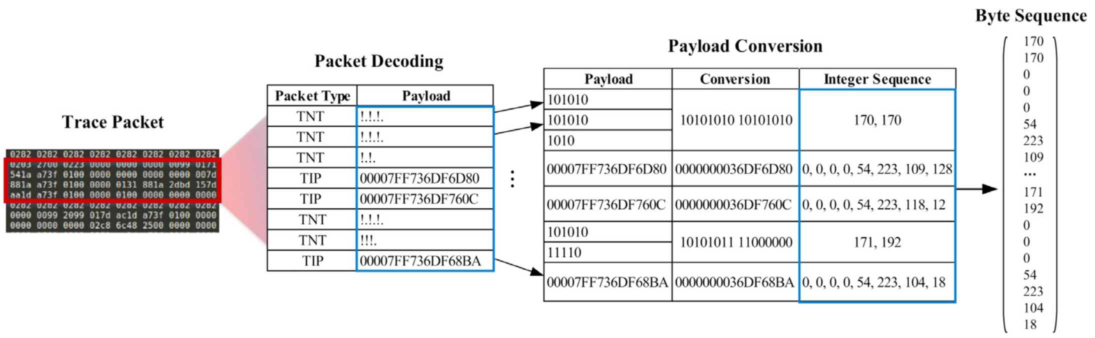

**Fig. 1.** Conversion of TNT and TIP packets into the behavioral byte sequence.

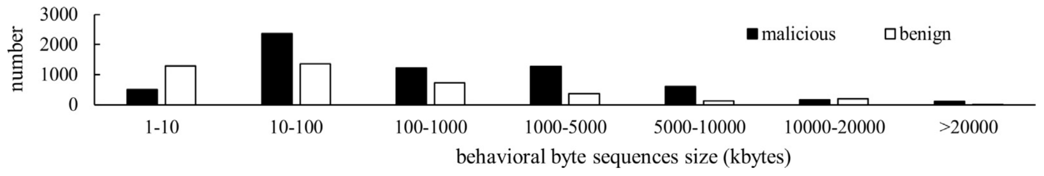

**Fig. 2.** Length distribution of behavioral byte sequences.

cessed. Finally, an 8-byte sequence is generated from every 64-bit offset.

**Byte sequence scaling.** According to the above method, TNT and TIP packets are converted into corresponding 1 or 8-byte data, respectively. Figure 1 shows an example of trace packets conversion. As a result, the execution trace data of each program can be expressed as a one-dimensional byte sequence $X_M$. As we know, the length of the byte sequence of different programs is very different. However, the classifier we designed needs to process fixed-length input vectors. Therefore, we follow Equation (1) and (2) to convert each sequence $(x_1, x_2, x_3, \cdots, x_M)$, to $(d_1, d_2, d_3, \cdots, d_L)$:

$$k_i = k_{i-1} + \left\lfloor \frac{M+i-1}{L} \right\rfloor (1 \le i \le L) \tag{1}$$

$$d_j = \frac{1}{k_j - k_{j-1}} \sum_{p=k_{j-1}+1}^{k_j} x_p (1 \le j \le L) \tag{2}$$

where $d_j$ is the element of the new sequence, $M$ is the length of the input sequence, and $L$ is a fixed size. The formula divides the input into $L$ sub-windows and uses the mean to form the output. The above formula is used to fix all sequences to the same size, and various length schemes have been attempted in the experiment, with a maximum length of 2 million.

### 3.2. Models for malware detection

Figure 2 shows the length distribution of one-dimensional byte sequences of all experimental samples. As shown in the figure, the length of feature sequences varies greatly, some of which are extremely long, exceeding 20 million. As we know, *long short term memory* (LSTM) (Hochreiter and Schmidhuber, 1997) is often used to process variable-length sequences. However, the extremely long feature sequence data makes it difficult for GPUs to provide enough memory for training, so learning from the original ultra-long sequences is a difficult task.

An intuitive method is to convert the sequences into images, that is, to transform malware classification into an image classification task. Chen et al. proposed HeNet (Chen et al., 2018), a neural network model based on control flow traces of program execution by transfer learning on Inception (Szegedy et al., 2017) and VGG (Simonyan and Zisserman, 2014). In the work of HeNet, Chen et al. (2018) split the control flow traces into $n$ segments of the same length. However, malicious programs contain common parts that both benign and malicious programs need to execute, so it is difficult to accurately segment and label their control flow traces. Using these disordered segments with inaccurate labels for training may lead to overfitting and reduce the generalization ability of the model. Moreover, when folding a one-dimensional sequence into a two-dimensional image, the spatially discontinuous pixels in the one-dimensional sequence will be calculated simultaneously for each sub-image processed by the two-dimensional convolution kernel. Therefore, the convolution of behavioral feature images will introduce non-existent spatial correlations, which is a false prior.

CNN has been proven to be suitable for byte sequence features in malware classification (Krčál et al., 2018; Raff et al., 2018). Therefore, we applied an 1D (one-dimensional) convolutional layer, which helps the model learn the correlation of program execution between local contexts from the whole sequence. Unlike raw bytes in binaries, there is a certain temporal semantic correlation between the local contexts of program behavior traces, so we tried to combine LSTM to further learn the temporal feature in the sequence. Compared with traditional *recurrent neural networks* (RNN), the adaptive gate in LSTM can make the memory unit have long-term memory capacity without losing short-term memory capacity, and can solve the vanishing gradient problem in RNN. The combination of CNN and LSTM has been proven to be very suitable to learn higher-level features in the input with sequential relationship (Alsulami and Mancoridis, 2018; Zhou et al., 2015).

Figure 3 shows the block diagram of the proposed models CNN-LSTM and CNN-BiLSTM, both of which consist of CNN and LSTM, but are different in unidirectional and bidirectional LSTM. First, an embedding layer is used to embed each byte of the sequence into a fixed-length vector, and the embedding layer is trainable during training. Then we applied an 1D convolutional layer to learn the correlation of program execution between local contexts from the whole sequence.

Before performing the LSTM, the convolutional activation is combined with the max-pooling layer to enable the model to generate activations regardless of the location of malicious features in the behavior sequence. This can accurately capture these sparse features from noisy behavior data, no matter how malicious and benign behaviors are mixed together. Following the previous layer, the LSTM (or BiLSTM) layer is used to learn high-level temporal information between local contexts.

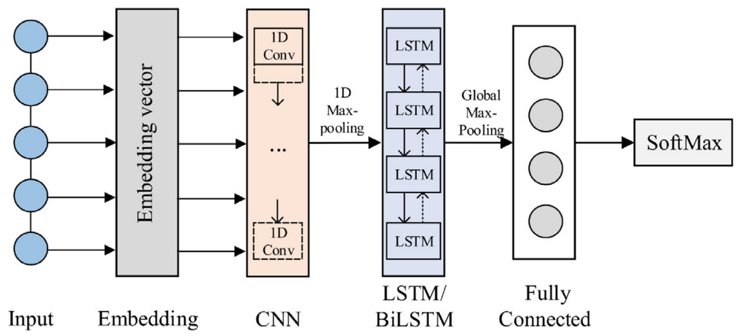

**Fig. 3.** The architecture diagram of CNN-LSTM and CNN-BiLSTM.

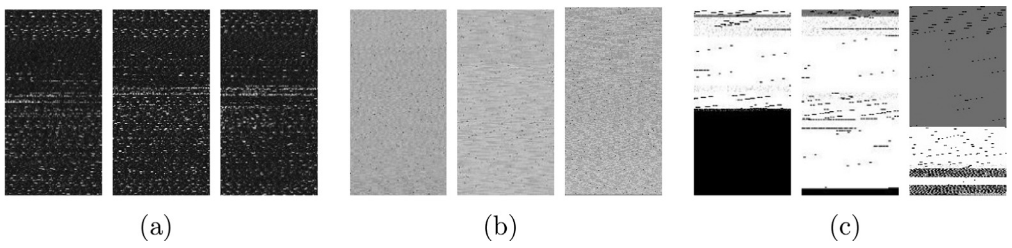

**Fig. 4.** Trace images of four malware variants: (a) Neshta, (b) Sality, (c) Expiro.

The hidden layer vector sequence output by the LSTM layer is then used as the input of the global max-pooling layer, which reduces the dimension of the output vector of the LSTM and outputs a global maximum vector as the final representation of the input sequence. The last layers of the model are the fully connected layer with the Softmax function to predict the probability. During the experiment, we have tried to stack LSTM layers, but the performance improvement is negligible. Considering the memory and time overhead, we ended up using only one LSTM (BiLSTM) layer.

### 3.3. Comparison models

In this section, various models used for comparison in the experiment are introduced. The proposed CNN-(Bi)LSTM is compared with the state-of-the-art models and other common models to check whether the model we designed can achieve better results.

Chen et al. (2018) proposed to classify malware based on the visual similarity between benign or malware control flow trace images. Figure 4 shows images of some malware families converted from control flow traces. As can be seen from the images, there are similarities in execution patterns of the same type of malware. We did not choose the image method of 2D convolution because the conversion introduces two new hyperparameters, the height and width of the image, which will become a new task to determine their size. At the same time, convolution on the transformed image of the behavior sequence will introduce non-existent spatial correlation. We used a similar method here, using 1D convolutional network architecture to learn the latent patterns of software execution from the original byte sequence.

First, we tried to use a relatively shallow network for comparison to see if our deep network can have better performance. Due to the limitation of the long input sequence, common artificial neural networks will suffer from insufficient memory errors. Inspired by the Raff et al. (2018) model, we applied the GCNN (Gated Convolutional Network) model based on the gated convolutional structure, because our input sequence has similar characteristics to the input of MalConv, namely, long sequence, mixture of malicious and benign features, and spatial correlation. Combining convolution activations with a global max-pooling, allows the model to detect features and generate activations regardless of the location. Second, we also tried a deeper ConvNet model, in which convolutional layers are stacked, and then the output is connected to the fully connected layer. Softmax activation is used to classify each input. Figure 5 shows the structure of GCNN and ConvNet, respectively.

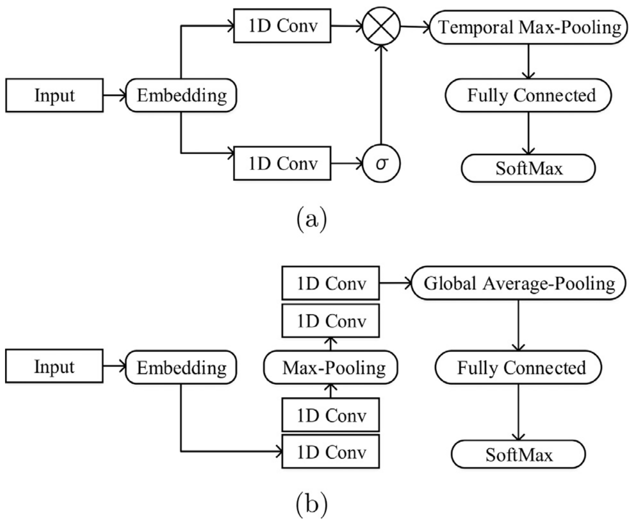

**Fig. 5.** Architecture diagrams of (a) GCNN and (b) ConvNet.

## 4. Dataset

The samples used in the experiments were collected from VirusTotal, clean Windows installations, and some free software sources. All malicious samples in our dataset obtained from VirusTotal contain a variety of malware families (e.g. worms, backdoors, trojans). These samples are detected in the wild, which can represent real malicious samples on the Internet. We obtained credible samples from a clean, fresh installed Windows 10 system, which contains commonly used applications and system programs. Since most of these programs are provided by Microsoft, in order to avoid the possible effect mentioned in the work of

**Table 2**
Category distribution of samples in the traning and test dataset.

| Section | Family | Number | Total |
|---|---|---|---|
| Malicious | Trojan | 3751 | 6109 |
| | virus | 754 | |
| | worm | 1000 | |
| | adware | 480 | |
| | backdoor | 124 | |
| Benign | - | 3961 | 3961 |

Raff et al. (2018), that is, to identify programs as “from Microsoft”, rather than as “benign”, we downloaded Windows executables from some free software sources (including PortableApp[1], Scoop[2] and SourceForge[3]).

In order to accurately determine the ground truth of all samples, we used VirusTotal (an online malicious file scanning service that contains more than 60 anti-virus engines) to scan each sample. For each uploaded file, VirusTotal provides a JSON report that contains the classification results of all engines. We analyzed these reports and used the method adopted in (Rosenberg et al., 2018, 2020) to determine the ground truth, which is described as follows. Samples that are not marked as malicious by any engine are labeled as benign, while those samples marked as malicious by more than 15 engines are labeled as malicious. For those samples that are marked as malicious by 0 to 15 engines, because their labels are controversial and may contaminate the dataset, we ignore these samples.

Because security vendors’ labeling formats on VirusTotal are inconsistent Alsulami and Mancoridis (2018); Raiu (2002), it is difficult to guarantee the accuracy of the labeling. In order to further capture the relations between multiple labels in the report of each sample and provide quantitative accuracy, we used the AVCLASS2 (Sebastián and Caballero, 2020) tool to automatically analyze JSON reports to provide clean classification labels.

It should be noted that some malicious samples will detect whether they are being monitored in sandboxes such as Cuckoo Sandbox to hide malicious behavior. Therefore, we used *YARA* rules[4] to exclude those samples with environmental detection behaviors. The final dataset contains 6,109 malicious samples and 3,961 benign samples (10,070 in total). Table 2 shows the category distribution of the final dataset. In order to measure the robustness of our approach against concept drift that may occur in real application situations, we further divided the dataset into training set *Dataset A* and test set *Dataset B* according to the timestamp when the sample first appeared in the VirusTotal analysis report. *Dataset A* is used for cross-validation experiments, which contains 8,906 samples that appeared from 2017-10-20 to 2018-6-19. *Dataset B* is used for the test, which contains 1,164 samples that appeared in the next four months after 2018-6-19. *Dataset A* and *DatasetB* have the same proportion of benign and malicious samples.

We further collected a total of 2,016 relatively newer malware scanned in 2019 and 2020 from VirusTotal to evaluate the adaptability of our method to the concept drift caused by the evolution of malware. These samples consist of four parts, namely, *Dataset 2019F, Dataset 2019S, Dataset 2020F*, and *Dataset 2020S*, which include 506, 506, 502 and 502 malware that appeared in the first half of 2019, the second half of 2019, the first half of 2020, and the second half of 2020, respectively.

[1] https://portableapps.com/2020.
[2] https://github.com/lukesampson/scoop.
[3] https://sourceforge.net/2020.
[4] https://github.com/Yara-Rules/rules.

## 5. Evaluation

To evaluate the efficiency and robustness of the proposed method, we seek answers to the following research questions:

**RQ1:** How effective can our method detect malware, i.e., What is the detection performance? How well does the method adapt to the concept drift caused by the evolution of malware? (Section 5.2)

**RQ2:** When faced with packing obfuscation, is the classifier based on API calls really more robust than that based on the static analysis? Does our method have better anti-interference ability than the classifier based on API calls? (Section 5.3)

**RQ3:** How robust is our method to detect adversarial samples? (Section 5.4)

### *5.1. Experiment protocol*

**Experiment setup.** To collect the trace of control flows of programs, we implemented a driver on the Windows 10 64-bit platform and used the official Intel decoder to decode the packets. We implemented all neural networks by using Pytorch Adam et al. (2016) and used four NVIDIA Tesla V100 GPU based on data parallelism to train the model.

**Hyper-parameters configuration.** The final hyper-parameters of the proposed model shown in Figure 3 are as follows. The embedding layer maps each byte in the input sequence into a unique 8-dimensional vector. The convolution kernel size of the 1D convolution layer is 64 and the stride is 16.

The next hidden LSTM layer has 128 units and the generated vector is sent to a global max-pooling layer. Then, the fully connected layer contains 128 neurons with the SELU activation function (Klambauer et al., 2017). Alternatively, if the BiLSTM layer is used instead of the LSTM layer, the fully connected layer will contain 256 neurons. The model finally uses the Softmax layer to classify hidden representations. The batch in experiments is set to 128 and the epoch in the ten-fold cross-validation is 15.

The weight of the network is modified by using SGD (Stochastic Gradient Descent) optimizer and momentum is set to speed up the model training. Weight decay and dropout between layers are used to avoid overfitting and thus improve generalization.

For the GCNN model, the size of the convolution kernel and the stride are both 500. For the ConvNet model, kernel sizes of 64 and 32 are applied to the first and second convolution block, and the corresponding stride sizes are 8 and 16, respectively. The number of neurons in the following fully connected layers are 192, 128, and 2, respectively.

**Metrics.** Accuracy, Area Under the Curve (AUC), True Positive Rate (TPR), False Positive Rate (FPR), and F1-score are used as standard metrics to evaluate the performance and the robustness. The training time indicates the average time taken to train the classifier, and the test time indicates the average overall time to identify the malware in the test dataset. Table 3 lists the description of the metrics used in the experiments.

### *5.2. Malware classification performance*

To answer research question RQ1, we measured the performance of the proposed method on the one hand, and compared it with other methods on the other hand.

In our experiments, we trained four models (CNN-LSTM, CNN-BiLSTM, GCNN, and ConvNet) on the control flow sequences. For each model, five rounds of 10-fold cross-validation training were performed and averaged to obtain fairer and more accurate results. In each 10-fold cross-validation, *Dataset A* was randomly shuffled. After determining the optimal hyper-parameters, we trained these models using all samples in *Dataset A*, and then tested them on

**Table 3**
Description of metrics in evaluation.

| Metrics | Description |
|---|---|
| True Positive(TP) | samples correctly labeled as malicious |
| True Negative(TN) | samples correctly labeled as benign |
| False Positive(FP) | samples incorrectly labeled as malicious |
| False Negative(FN) | samples incorrectly labeled as benign |
| True Positive Rate(TPR) | TP/(TP+FN) |
| False Positive Rate(FPR) | FP/(FP+TN) |
| Accuracy(Acc) | (TP+TN)/(TP+TN+FP+FN) |
| F1-score(F1) | 2*Precision*Recall/(Precision+Recall) |
| Area Under the Curve (AUC) | the area under the ROC curve |
| Training time | the average time to train the classifier |
| Test time | the average overall time to identify the malware |
| MTTD | mean time to detect |

**Table 4**
Performance of different models using 10-fold cross-validation on *Dataset A*.

| Model | 500,000 | | | | |
|---|---|---|---|---|---|
| | Acc | TPR | FPR | F1-score | Training time (minutes) |
| GCNN | 0.915 | 0.947 | 0.135 | 0.931 | 4.1 |
| ConvNet | 0.919 | 0.944 | 0.117 | 0.935 | 7.6 |
| CNN-BiLSTM | 0.946 | 0.959 | 0.072 | 0.957 | 11.3 |
| CNN-LSTM | 0.942 | 0.962 | 0.088 | 0.953 | 11.2 |
| | | | **1,000,000** | | |
| | Acc | TPR | FPR | F1-score | Training time (minutes) |
| GCNN | 0.919 | 0.946 | 0.124 | 0.934 | 8.7 |
| ConvNet | 0.930 | 0.949 | 0.099 | 0.943 | 13.5 |
| CNN-BiLSTM | 0.951 | 0.965 | 0.071 | 0.960 | 21.6 |
| CNN-LSTM | 0.949 | 0.964 | 0.074 | 0.959 | 18.6 |
| | | | **1,500,000** | | |
| | Acc | TPR | FPR | F1-score | Training time (minutes) |
| GCNN | 0.921 | 0.952 | 0.127 | 0.936 | 10.9 |
| ConvNet | 0.929 | 0.949 | 0.101 | 0.943 | 16.1 |
| CNN-BiLSTM | 0.954 | 0.967 | 0.066 | 0.963 | 28.1 |
| CNN-LSTM | 0.951 | 0.966 | 0.072 | 0.960 | 23.4 |
| | | | **2,000,000** | | |
| | Acc | TPR | FPR | F1-score | Training time (minutes) |
| GCNN | 0.922 | 0.955 | 0.131 | 0.937 | 13.2 |
| ConvNet | 0.927 | 0.948 | 0.106 | 0.941 | 18.7 |
| CNN-BiLSTM | **0.957** | **0.969** | **0.063** | **0.964** | 34.5 |
| CNN-LSTM | 0.952 | 0.968 | 0.071 | 0.962 | 28.2 |

*Dataset B* to measure their generalization ability to detect unseen samples.

**Experiment I: detection effectiveness.** Table 4 and Table 5 show the performance changes of various models as the length increases when performing 10-fold cross-validation on *Dataset A* and testing on *Dataset B*, respectively.

The two tables also show the time spent on training and test for different models. It can be seen that the proposed methods take more time than the other methods, but still in the same order of magnitude, i.e., only two to three times as much quantitatively. It is worth mentioning that test time is the average overall time to identify all malware in the test set, and for a particular malware, its MTTD (Mean time to detect) should be this overall time divided by the number of samples in the test set (i.e., 1164 in our experimental setup). It can also be seen that all MTTDs are in the order of milliseconds, with the maximum value of 17.01 ms for CNN-BiLSTM when the input length is 2 million.

Figure 6 shows the corresponding ROC curve for the testing result on *Dataset B*.

It can be seen that as the length of the input sequence increases to make the data more effective, the performance of all models is improved. For cross-validation, among all models, CNN-BiLSTM has the highest accuracy under any input length, reaching 95.7% when the input length is 2 million, and the corresponding F1-score is 0.964 (see Table 4). For the test results shown in Table 5, as the length increases, the gain obtained is not as large as the gain obtained in cross-validation. After the length is 1 million, the accuracy of CNN-BiLSTM on unseen samples only increases by 0.1% at a time. CNN-LSTM also shows the same results, with an increase of 0.2% each time. At the same time, ConvNet shows slight overfitting. Namely, combining its cross-validation and testing performance, when the input length is 2 million, the accuracy and TPR decrease by 4.4% (from 92.7% to 88.3%) and 7.4% (from 94.8% to 87.4%), respectively. Compared with other models, CNN-LSTM and CNN-BiLSTM have the best and very close performance in testing, with almost equal F1-score of 0.956 and 0.954 (see Table 5), almost equal AUC of 0.991 (see Figure ), and the corresponding accuracy of 94.6% and 94.5%. Meanwhile, compared to using LSTM, BiLSTM does not bring significant performance improvement. From the above results, it can be seen that combining CNN and LSTM is more suitable for learning from control flow traces.

**Experiment II: adaptability to newer malware.** In order to test the adaptability of the proposed classifier to the evolution of malicious samples, we used newer malicious samples that appeared in 2019 and 2020 to conduct experiments. These newer samples are divided into four test sets based on the timestamp (Section 4). We trained each model by using *Dataset A*, and then conducted classification on these test sets that include newer malicious samples.

Figure 7 shows the accuracy trend of the four models in detecting new samples under different input lengths.

When tested on *Dataset 2019F*, all classifiers can achieve relatively optimal performance. This also shows that when the time

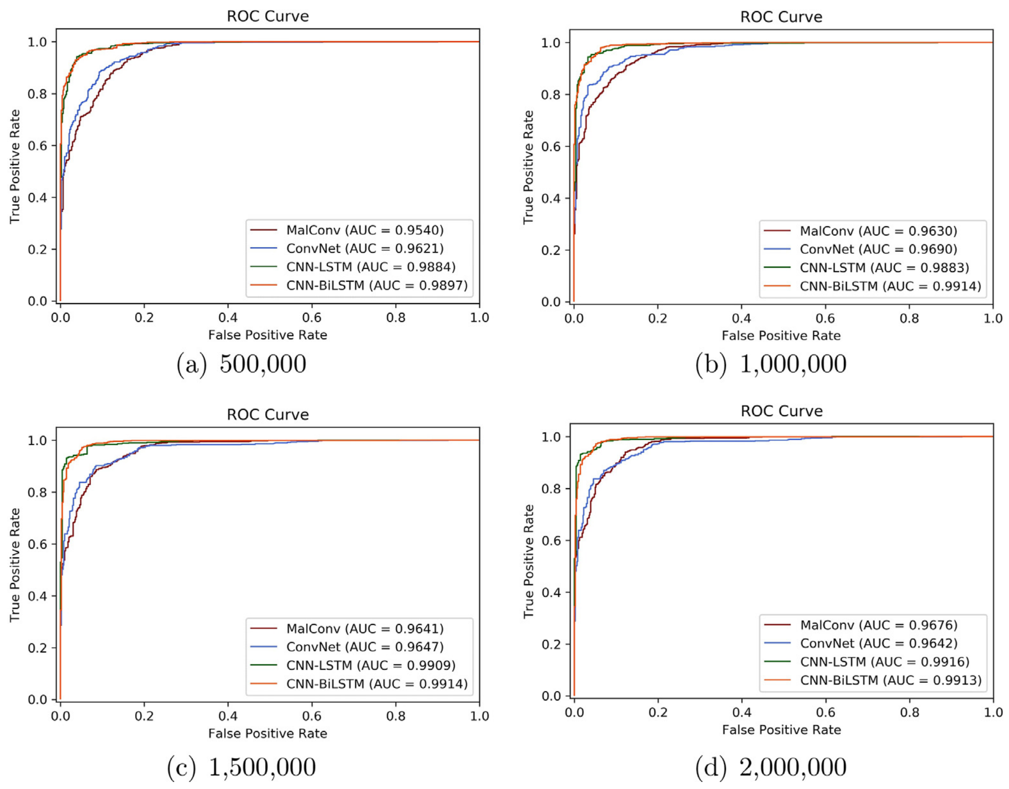

**Fig. 6.** ROC curves of different models with various input lengths when testing on *Dataset B*.

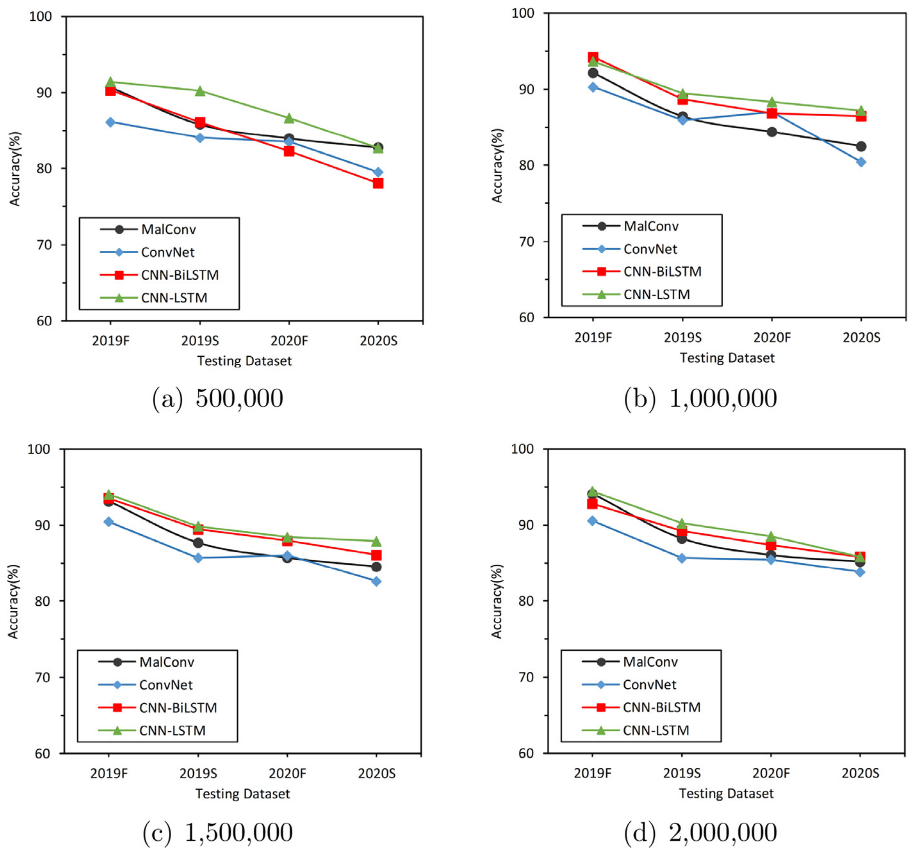

**Fig. 7.** The accuracy of different models tested on newer samples.

**Table 5**
Performance of different models tested on *Dataset B* with *Dataset A* for training.

| Model | 500,000 | | | | | |
|---|---|---|---|---|---|---|
| | Acc | TPR | FPR | F1-score | Test time (minutes) | MTTD (milliseconds) |
| GCNN | 0.875 | 0.882 | 0.135 | 0.896 | 0.05 | 2.58 |
| ConvNet | 0.877 | 0.868 | 0.111 | 0.895 | 0.07 | 3.61 |
| CNN-BiLSTM | 0.933 | 0.93 | 0.062 | 0.944 | 0.10 | 5.15 |
| CNN-LSTM | 0.939 | 0.942 | 0.066 | 0.949 | 0.10 | 5.15 |
| | | | **1,000,000** | | | |
| | Acc | TPR | FPR | F1-score | Test time (minutes) | MTTD (milliseconds) |
| GCNN | 0.884 | 0.888 | 0.123 | 0.903 | 0.10 | 5.15 |
| ConvNet | 0.894 | 0.881 | 0.087 | 0.910 | 0.12 | 6.19 |
| CNN-BiLSTM | 0.943 | 0.947 | 0.062 | 0.953 | 0.24 | 12.37 |
| CNN-LSTM | 0.941 | 0.938 | 0.055 | 0.951 | 0.20 | 10.31 |
| | | | **1,500,000** | | | |
| | Acc | TPR | FPR | F1-score | Test time (minutes) | MTTD (milliseconds) |
| GCNN | 0.893 | 0.906 | 0.126 | 0.912 | 0.11 | 5.67 |
| ConvNet | 0.889 | 0.876 | 0.089 | 0.906 | 0.15 | 7.73 |
| CNN-BiLSTM | 0.944 | 0.943 | 0.056 | 0.953 | 0.31 | 15.98 |
| CNN-LSTM | 0.944 | 0.942 | 0.054 | 0.953 | 0.26 | 13.40 |
| | | | **2,000,000** | | | |
| | Acc | TPR | FPR | F1-score | Test time (minutes) | MTTD (milliseconds) |
| GCNN | 0.900 | 0.919 | 0.125 | 0.919 | 0.13 | 6.70 |
| ConvNet | 0.883 | 0.874 | 0.106 | 0.900 | 0.18 | 9.28 |
| CNN-BiLSTM | **0.945** | 0.942 | **0.051** | 0.954 | 0.33 | **17.01** |
| CNN-LSTM | **0.946** | **0.946** | 0.053 | **0.956** | 0.29 | 14.95 |

interval between the training and test sets does not exceed one year (from 2018-6 to 2019-6), these classifiers can detect malware with high accuracy and show good performance close to that in *Experiment I*. The accuracy decreases as the time interval increases. Moreover, when the length of the input sequence is short, the accuracy of all models decrease more significantly. As the input length increases, the accuracy improves and becomes more stable on different test sets. CNN-LSTM tends to perform better than other models on different test cases and can better adapt to the evolution of malicious samples (due to more stable accuracy), followed by CNN-BiLSTM.

**Experiment III: performance comparison with other dynamic methods.** We used CNN-BiLSTM and CNN-LSTM with an input length of 2 million to compare with two common types of dynamic methods in related work, namely, *Neurlux* described in the work of Jindal et al. (2019), and the method described in the work of Kolosnjaji et al. (2016) (which we will call *LSTMAPI*). *Neurlux* uses dynamic analysis reports collected from the sandbox as input, and uses a combination of CNN and LSTM for classification. The report records the detailed behavioral information generated by the program, and *Neurlux* uses the word sequence present in the report to predict whether the report comes from a malicious binary file. *LSTMAPI* is a typical malware detection method that uses system call sequence features for deep learning and the sandbox to trace API call sequences. In this comparative experiment, we used the Cuckoo sandbox to trace all programs to obtain dynamic analysis reports. These reports were directly used to train *Neurlux*. On the other hand, for *LSTMAPI*, the API call sequences in these reports were encoded as one-hot vector and the LSTM layer were used to output malicious or benign classification result.

Table 6 shows the performance of different dynamic methods tested using 10-fold validation on *Dataset A*, and Table 7 shows the results tested on *Dataset B*. From the results, we can see that our methods perform relatively well, where CNN-BiLSTM has the best accuracy of 95.7% on *Dataset A*, and CNN-LSTM has the best accuracy of 94.6% on *Dataset B*. Although Neurlux performs slightly better than CNN-LSTM in 10-fold cross-validation, it is not robust enough to the newer samples in *Dataset B*.

On the other hand, we designed experiments to compare the performance of our method with HeNet Chen et al. (2018), a malware detection method that is also based on control flow traces.

**Table 6**
Performance of different dynamic methods using 10-fold validation on *Dataset A*.

| | Accuracy | TPR | FPR | F1-score |
|---|---|---|---|---|
| LSTMAPI | 0.927 | 0.935 | 0.087 | 0.935 |
| Neurlux | 0.955 | 0.963 | 0.058 | 0.963 |
| CNN-LSTM | 0.952 | 0.968 | 0.071 | 0.962 |
| CNN-BiLSTM | **0.957** | 0.969 | 0.063 | **0.964** |

**Table 7**
Performance of different dynamic methods tested on *Dataset B* with *Dataset A* for training.

| | Accuracy | TPR | FPR | F1-score |
|---|---|---|---|---|
| LSTMAPI | 0.918 | 0.913 | 0.062 | 0.925 |
| Neurlux | 0.934 | 0.931 | 0.048 | 0.940 |
| CNN-LSTM | **0.946** | 0.946 | 0.053 | **0.956** |
| CNN-BiLSTM | 0.945 | 0.942 | 0.051 | 0.954 |

**Table 8**
Test results for ROP attacks.

| | Accuracy | FPR | AUC |
|---|---|---|---|
| **CNN-BiLSTM** | **0.99** | **0.0011** | **0.998** |
| **CNN-LSTM** | 0.99 | 0.0032 | 0.997 |
| **HeNet** | 0.98 | 0.0073 | 0.989 |

Since only ROP attacks against Adobe Reader were detected in HeNet, we also evaluated ROP attacks for comparison. We collected 287 malicious and 360 benign PDF samples of a similar size to the dataset evaluated by HeNet. We also applied 10-fold cross-validation for evaluation, which is the same as the experimental setup of HeNet.

Table 8 shows the performance comparison. On the test set, CNN-BiLSTM has an accuracy of 99% and an AUC of 0.998, outperforming HeNet.

Moreover, in cross-validation experiments, our model can usually obtain an accuracy of 100% in a certain round. This is not surprising because ROP attacks are typical attacks against control flows, and the malicious patterns in the trace sequences are relatively obvious, so there is no big challenge to our classifier. It is worth mentioning that in HeNet, with only 647 samples, more

**Table 9**
The baseline performance tested on *Dataset B*.

| | Accuracy | TPR | FPR |
|---|---|---|---|
| MalConv:BI | 0.909 | 0.892 | 0.054 |
| LSTM:API | 0.918 | 0.913 | 0.062 |
| CNN-BiLSTM:CF | 0.945 | 0.942 | 0.051 |

than 700,000 trace images were generated, requiring a lot of tedious manual work to determine their ground truth. In contrast, our method is simpler and does not require tedious manual analysis, while showing better performance.

## 5.3. Impact of packing

To answer research question RQ2, we evaluated the robustness of the proposed classifier when the packing distribution was uneven, by comparing our method using the best-performing configuration with a static analysis method and a dynamic analysis method.

We constructed a series of sub-datasets with different ratios of packed programs to perform the experiments. We used the "malicious packed ratio" to refer to the percentage of packed samples among malicious samples. We used the "benign packed ratio" to refer to the percentage of packed samples among benign samples. In order to generate these subsets, it is needed to detect whether a sample is packed. Therefore, we used *Exeinfo-PE, PEiD, YARA* rules, and *DIE*[5] to identify packed samples in *Dataset A* (for comparison purposes, files larger than 2M were filtered out, since MalConv can only detect files smaller than 2MB). In *Dataset A*, out of 4,837 malicious binaries, 1,773 (36.7%) are packed, and out of 2,704 benign binaries, 111 (4.1%) are packed. We randomly selected 1,662 benign and 1,662 malicious samples from the unpacked part to form *Dataset P*. Then, by replacing the unpacked malicious samples in *Dataset P* with packed malware, we increased the "malicious packed ratio' from 0 to 1, with a step size of 10%, and we obtained 11 sub-datasets for training. Similarly, we increased the "benign packed ratio" by replacing the unpacked benign samples in *Dataset P* with manually packed benign samples. Finally, we obtained another 11 sub-datasets with different "benign packed ratio". For all training cases, we tested all classifiers on the same test set (for both benign samples and malicious samples, the ratio of packed samples to unpacked samples is 1:1).

In the experiments, we measured the performance of CNN-BiLSTM:CF (on control flow traces), MalConv:BI (Raff et al., 2018) (on static binaries), and LSTM:API (on API call sequences). The MalConv:BI classifier is a representative model based on static analysis (Raff et al., 2018), which uses the raw byte sequence of binaries as input and is trained on a gated convolutional network. We used the binaries in our dataset to train MalConv:BI. LSTM:API (called LSTMAPI in Section 5.2) is a deep learning method based on the sequence of system API calls. We executed each sample in the Cuckoo sandbox, and then extracted the API call sequence from the report. The classifier uses LSTM layer to learn these feature sequences and output prediction results.

Before studying the impact of packing, we trained and tested the three models without considering whether each sample is packed and how the packing is distributed. For CNN-BiLSTM, we used the best performance configuration with an input length of 2 million. Table 9 shows the results of these models tested on *Dataset B*. We can conclude that the performance of the three methods is relatively good without considering packing, that is, the accuracy is more than 90%, and the TPR is large (90-92%), and the FPR is small (5-6%).

**Experiment I: various malicious packed ratios.** Figure 8 shows the TPR and FPR trends of these classifiers on the test set as the malicious packed ratio in the training set increases. As the malicious packed ratio increases from 0% to 100%, the static classifier (MalConv:BI) keeps a high TPR (for packed malicious samples, see Figure ), and a very low FPR (for unpacked benign samples, see Figure ). However, for the unpacked malicious samples, the TPR decreases from 88.3% to 55.5%. Moreover, for the packed benign samples, the FPR is very high (even up to 89%), which means that the classifier almost loses the malware detection ability on the packed test set. The experimental results indicate that if the training dataset does not contain packed benign samples, the static model trained by only using packed malicious samples will treat the packing attribute as the malicious feature, resulting in a poor testing performance for both unpacked malicious samples and packed benign samples.

The lack of overlap between packed benign and malicious samples will make the classifier more inclined to detect whether the software is packed than whether it is malicious, similar to the finding of (Aghakhani et al., 2020).

It is generally believed that API call-based classifiers are more robust to packing, but our experimental results show that their performance is actually unsatisfactory.

As shown in Figure 8, as the malicious packed ratio increases, the LSTM:API maintains a high TPR on packed samples, but does not show the expected robustness on unpacked samples. Its TPR on unpacked samples rapidly decreases from 85.9% to 40.9% (see Figure ), which is even much worse than MalConv:BI (55.5%), thus bringing greater potential risk than the static classifier (MalConv:BI). This poses more potential risk and is more unacceptable than misclassifying benign samples as malicious samples. We conclude that compared with the static method, the API call-based method does show better robustness to packed samples (see FPR in Figure ), but it also shows worse robustness to unpacked samples (see TPR in Figure ).

In contrast, CNN-BiLSTM:CF outperforms LSTM:API in every case. For the test of packed samples, when the malicious packed ratio is 0, CNN-BiLSTM:CF can provide a low FPR of 8.4%, while LSTM:API provides an FPR of 16.7% (see Figure ). Even when all unpacked malicious samples are replaced, the FPR of CNN-BiLSTM:CF increases by only 9%, while the FPR of LSTM:API increase by 12.6% (see Figure ). For the test of unpacked samples, CNN-BiLSTM:CF also outperforms LSTM:API. In particular, when the malicious packed ratio is 1, CNN-BiLSTM:CF gives a higher TPR value than LSTM:API by about 26.3% (67.2% minus 40.9%), as shown in Figure.

We can conclude that the API call-based method is not robust enough in the face of packing interference in malware, while our proposed method based on the control flow traces is more robust, especially when testing unpacked samples.

**Experiment II: various benign packed ratios.** Figure 9 shows the TPR and FPR trends of these classifiers on the test set as the benign packed ratio in the training set increases. The increase of benign packed ratio helps the static classifier (MalConv:BI) to reduce the FPR on packed samples and maintain a high TPR (see Figure ). However, for unpacked samples, it cannot achieve good performance. Although the FPR is very low, the TPR decreases from 55.3% to 38.6% (see Figure ). This suggests that if all samples in training dataset are packed, the static classifier cannot achieve good performance on the unpacked dataset, because what the model learned in packed samples is not enough to deal with unpacked samples. This conclusion is consistent with that in (Aghakhani et al., 2020).

[5] https://github.com/horsicq/Detect-It-Easy.

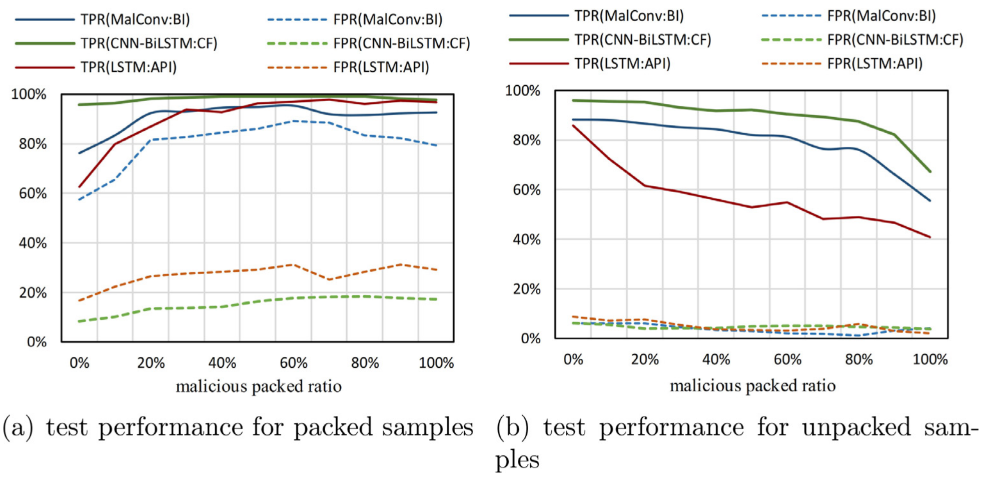

**Fig. 8.** Test performance of different models under a series of training datasets with different malicious packed ratios.

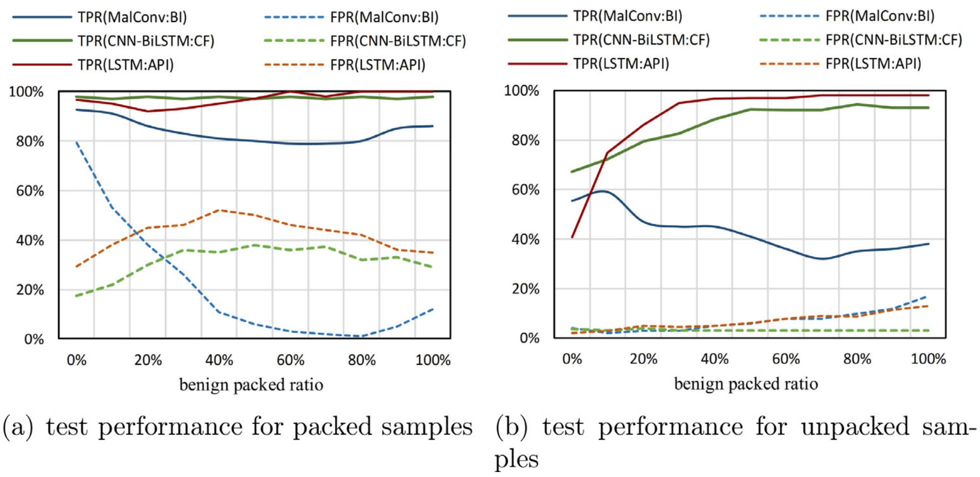

**Fig. 9.** Test performance of different models under a series of training datasets with different benign packed ratios.

In this experiment, LSTM:API shows better robustness than that in *Experiment I*. As shown in Figure 9, with the increase of benign packed ratio, the LSTM:API maintains a high TPR on packed samples, while its TPR on unpacked samples also increases rapidly. In contrast, CNN-BiLSTM:CF still can provide a lower FPR (maximum FPR of 38%) than that of LSTM:API (maximum FPR of 52%) on packed samples, and also performs well on unpacked samples. Compared with MalConv:BI, two dynamic classifiers, LSTM:API and CNN-BiLSTM:CF achieve higher FPRs on packed samples, but show better robustness on unpacked samples. The results indicate that using only packed samples as the training dataset also has a significant impact on the dynamic classifier, making the classification results misleading.

**Experiment III: "packed" vs "anti-virus engine".** It is worth noting that using VirusTotal for labeling will lead to a high proportion of packed malware and a low proportion of packed benign samples. To verify this, we packed 500 unpacked benign programs with commonly used packers including UPX, Simple Packer, AsPack, and MPRESS, and then uploaded these packed programs to VirusTotal. The results of virus scan showed that only 12% of the samples were not labeled as malicious by any engine, while more than 44% were labeled as malicious by at least 5 engines, indicating that anti-virus engines on VirusTotal prefer to label packed files as malicious. According to label filtering strategy used in Section 4, this explains why the proportion of packed benign samples in our dataset is much smaller than that in real-world datasets (Aghakhani et al., 2020; Rahbarinia et al., 2017).

The above experiments show that in the classification of malicious programs based on deep learning, not only static analysis methods, but also dynamic analysis methods will learn the packing attributes as malicious features, especially when the packed ratio of malicious and benign programs deviates significantly. At the same time, labeling based on VirusTotal will lead to a large deviation in the packed ratio of benign and malicious samples. In these highly deviated datasets, it will be easier for dynamic analysis methods with poor robustness to achieve overall high test performance. However, when packed and unpacked samples are tested separately, the vulnerability of these methods will be exposed, resulting in a high rate of misclassification. This is a significant issue because malware detection studies often rely on VirusTotal's services to establish the ground-truth of whether a sample is benign or malicious (Copty et al., 2018; Rathore et al., 2018; Rhode et al., 2018). In short, we prove that the method proposed in this paper is more resistant to packing interference.

### 5.4. Robustness to adversarial samples

To answer research question RQ3, we implemented a state-of-the-art adversarial attack to test the robustness of CNN-BiLSTM and CNN-LSTM against adversarial samples. In recent related work, Rosenberg et al. (2020) proposed a general black-box attack against API call-based deep learning malware classifiers, which is effective against various classifier architectures. According to their proposed algorithm, the attacker can modify the API call sequence in a ma-

**Table 10**
Performance of three models in detecting adversarial samples.

| | Succeeded Samples | Failed Samples |
|---|---|---|
| Classifier | Accuracy | Accuracy |
| LSTM:API | 0% | 100% |
| CNN-LSTM | 84% | 87% |
| CNN-BiLSTM | 89% | 81% |

licious binary to convert it into adversarial samples while maintaining the original functionality, thereby evading detection. As we know, the source code of malware is usually unavailable, so it is difficult to directly modify the control flow of the malware. However, through the above attack method, we can add API calls into the binary, thus adding control flow jump instructions to affect the original control flow of the program. We conducted this experiment to test whether such control flow changes generated by adding API calls can help malicious samples evade detection.

Our implementation of the experimental process can be divided into three phases, 1) creating a target API call-based classifier to perform the black-box attack, 2) generating adversarial samples against the target classifier by modifying the malware using the algorithm in (Rosenberg et al., 2020), 3) using adversarial samples to attack our classifier based on control flow traces.

Regarding the target classifier, we used the previously implemented LSTM:API classifier (see Section 5.3). Then, we followed the additional constraints proposed in the original work (Rosenberg et al., 2020) to generate adversarial samples against the API call-based classifier.

We finally generated adversarial samples which can successfully evade detection by the LSTM:API classifier. It is worth mentioning that all the malicious samples we used were correctly classified by three classifiers before being modified. Then, we traced control flow traces of these adversarial samples and extracted their behavioral byte sequences for detection. As shown in Table 10, 16% of these samples successfully attacked CNN-LSTM, i.e., evaded detection, while 11% successfully attacked CNN-BiLSTM. In addition, we tested those failed adversarial samples, that is, no matter how we modified them, we cannot make them escape detection by LSTM:API. Although these samples failed to escape, they may be effective for CNN-LSTM and CNN-BiLSTM.

The results show that for these failed adversarial samples, only 13% evaded detection by CNN-LSTM, and 19% evaded detection by CNN-BiLSTM. In summary, we conclude that our classifier (CNN-LSTM and CNN-BiLSTM) have better robustness against adversarial attacks than the classifier based on API calls.

## 6. Conclusion

In this work, we proposed an effective and robust deep learning based malware detection approach. We relied on the instruction-level control flow traces as the feature set and converted the traces into behavioral byte sequences as the input of the neural network without the need for complicated feature extraction. The proposed CNN-LSTM and CNN-BiLSTM architectures can capture high level information in the local contexts while preserving the semantic correlations between them. The evaluation results show that our approach can effectively detect unseen samples with an accuracy of 94.6% and has good adaptability to the evolution of malware. We also found that the method based on API call sequences is not as robust as commonly believed, while our method based on control flow traces is more robust under different malware packed ratios, especially when detecting unpacked samples.

In addition, the evaluation also shows that our method is more robust than the API call-based method in defending against adversarial attacks.

## Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

## References

Adam, P., Sam, G., Soumith, C., Gregory, C., 2016. Pytorch. https://github.com/pytorch/pytorch.

Aghakhani, H., Gritti, F., Mecca, F., Lindorfer, M., Ortolani, S., Balzarotti, D., Vigna, G., Kruegel, C., 2020. When malware is packin'heat; limits of machine learning classifiers based on static analysis features. In: Proceedings of the 27th Annual Network and Distributed System Security Symposium Network and Distributed Systems Security (NDSS 2020). The Internet Society.

Alsulami, B., Mancoridis, S., 2018. Behavioral malware classification using convolutional recurrent neural networks. In: Proceedings of the 13th International Conference on Malicious and Unwanted Software (MALWARE 2018 ). IEEE, pp. 103–111.

Anderson, H., 2019. Machine learning static evasion competition. https://www.elastic.co/cn/blog/machine-learning-static-evasion-competition.

Anderson, H.S., Roth, P., 2018. Ember: an open dataset for training static pe malware machine learning models. arXiv preprint arXiv:1804.04637.

Chen, L., Sultana, S., Sahita, R., 2018. Henet: A deep learning approach on intel® processor trace for effective exploit detection. In: Proceedings of the 2018 IEEE Security and Privacy Workshops (SPW 2018 ). IEEE, pp. 109–115.

Copty, F., Danos, M., Edelstein, O., Eisner, C., Murik, D., Zeltser, B., 2018. Accurate malware detection by extreme abstraction. In: Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC 2018). ACM, pp. 101–111.

David, O.E., Netanyahu, N.S., 2015. Deepsign: Deep learning for automatic malware signature generation and classification. In: Proceedings of the 2015 International Joint Conference on Neural Networks (IJCNN 2015). IEEE, pp. 1–8.

Ficco, M., 2022. Malware analysis by combining multiple detectors and observation windows. IEEE Transactions on Computers 71 (5), 1276–1290.

Firdausi, I., Erwin, A., Nugroho, A.S., et al., 2010. Analysis of machine learning techniques used in behavior-based malware detection. In: Proceedings of the Second International Conference on Advances in Computing, Control, and Telecommunication Technologies. IEEE, pp. 201–203.

Fleshman, W., 2019. Evading machine learning malware classifiers. https://www.mcafee.com/enterprise/en-us/assets/reports/rp-quarterly-threats-july-2020.pdf.

Fredrikson, M., Jha, S., Christodorescu, M., Sailer, R., Yan, X., 2010. Synthesizing near-optimal malware specifications from suspicious behaviors. In: Proceedings of the 31st IEEE Symposium on Security and Privacy (S&P 2010). IEEE, pp. 45–60.

Hardy, W., Chen, L., Hou, S., Ye, Y., Li, X., 2016. Dl4md: A deep learning framework for intelligent malware detection. In: Proceedings of the 2016 International Conference on Data Science (ICDATA). WorldComp.

Hochreiter, S., Schmidhuber, J., 1997. Long short-term memory. Neural computation 9 (8), 1735–1780.

Hu, W., Tan, Y., 2017. Black-box attacks against rnn based malware detection algorithms. arXiv preprint arXiv:1705.08131.

Hu, W., Tan, Y., 2017. Generating adversarial malware examples for black-box attacks based on gan. arXiv preprint arXiv:1702.05983

Huang, W., Stokes, J.W., 2016. Mtnet: a multi-task neural network for dynamic malware classification. In: Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA 2016). Springer, pp. 399–418.

Jindal, C., Salls, C., Aghakhani, H., Long, K., Kruegel, C., Vigna, G., 2019. Neurlux: dynamic malware analysis without feature engineering. In: Proceedings of the 35th Annual Computer Security Applications Conference (ACSAC 2019). ACM, pp. 444–455.

Kephart, J.O., Sorkin, G.B., Arnold, W.C., Chess, D.M., Tesauro, G.J., White, S.R., Watson, T., 1995. Biologically inspired defenses against computer viruses. In: IJCAI (1), pp. 985–996.

Klambauer, G., Unterthiner, T., Mayr, A., Hochreiter, S., 2017. Self-normalizing neural networks. arXiv preprint arXiv:1706.02515.

Kolosnjaji, B., Demontis, A., Biggio, B., Maiorca, D., Giacinto, G., Eckert, C., Roli, F., 2018. Adversarial malware binaries: Evading deep learning for malware detection in executables. In: Proceeding of the 26th European signal processing conference (EUSIPCO 2018 ). IEEE, pp. 533–537.

Kolosnjaji, B., Zarras, A., Webster, G., Eckert, C., 2016. Deep learning for classification of malware system call sequences. In: Proceedings of the 29th Australasian Joint Conference on Artificial Intelligence. Springer, pp. 137–149.

Krčál, M., Švec, O., Bálek, M., Jašek, O., 2018. Deep convolutional malware classifiers can learn from raw executables and labels only.

Le, Q., Boydell, O., Mac Namee, B., Scanlon, M., 2018. Deep learning at the shallow end: Malware classification for non-domain experts. Digital Investigation 26, S118–S126.

McAfee, 2021. Mcafee labs threats report. https://www.mcafee.com/enterprise/en-us/assets/reports/rp-threats-jun-2021.pdf.

Nataraj, L., Karthikeyan, S., Jacob, G., Manjunath, B.S., 2011. Malware images: visualization and automatic classification. In: Proceedings of the 8th international symposium on visualization for cyber security, pp. 1–7.

Nataraj, L., Yegneswaran, V., Porras, P., Zhang, J., 2011. A comparative assessment of malware classification using binary texture analysis and dynamic analysis. In: Proceedings of the 4th ACM Workshop on Security and Artificial Intelligence, pp. 21–30.

Raff, E., Barker, J., Sylvester, J., Brandon, R., Catanzaro, B., Nicholas, C., 2018. Malware detection by eating a whole exe.

Raff, E., Sylvester, J., Nicholas, C., 2017. Learning the pe header, malware detection with minimal domain knowledge. In: Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security, pp. 121–132.

Rahbarinia, B., Balduzzi, M., Perdisci, R., 2017. Exploring the long tail of (malicious) software downloads. In: Proceedings of the 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2017 ). IEEE, pp. 391–402.

Raiu, C., 2002. A virus by any other name: Virus naming practices. Security Focus.

Rathore, H., Agarwal, S., Sahay, S.K., Sewak, M., 2018. Malware detection using machine learning and deep learning. In: Proceedings of the 6th International Conference on Big Data Analytics (BDA 2018). Springer, pp. 402–411.

Rhode, M., Burnap, P., Jones, K., 2018. Early-stage malware prediction using recurrent neural networks. Computers & Security 77, 578–594.

Ronghua, T., Rafiqul, I., Lynn, B., Steve, V., 2010. Differentiating malware from cleanware using behavioural analysis. In: Proceedings of the 5th International Conference on Malicious and Unwanted Software (MALWARE 2010). IEEE, pp. 23–30.

Rosenberg, I., Shabtai, A., Elovici, Y., Rokach, L., 2020. Query-efficient black-box attack against sequence-based malware classifiers. In: Proceedings of the 36th Annual Computer Security Applications Conference (ACSAC 2020). ACM, pp. 611–626.

Rosenberg, I., Shabtai, A., Rokach, L., Elovici, Y., 2018. Generic black-box end-to-end attack against state of the art api call based malware classifiers. In: Proceedings of the 21st International Symposium on Research in Attacks, Intrusions, and Defenses ( RAID 2018). Springer, pp. 490–510.

Saxe, J., Berlin, K., 2015. Deep neural network based malware detection using two dimensional binary program features. In: Proceedings of the 10th International Conference on Malicious and Unwanted Software (MALWARE 2015). IEEE, pp. 11–20.

Sebastián, S., Caballero, J., 2020. Avclass2: Massive malware tag extraction from av labels. In: Proceedings of the 36th Annual Computer Security Applications Conference (ACSAC 2020). ACM, pp. 42–53.

Simonyan, K., Zisserman, A., 2014. Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556.

Szegedy, C., Ioffe, S., Vanhoucke, V., Alemi, A., 2017. Inception-v4, inception-resnet and the impact of residual connections on learning. In: Proceedings of the AAAI Conference on Artificial Intelligence, Vol. 31.

Tobiyama, S., Yamaguchi, Y., Shimada, H., Ikuse, T., Yagi, T., 2016. Malware detection with deep neural network using process behavior. In: Proceedings of the 40th IEEE Annual Computer Software and Applications Conference (COMPSAC 2016). IEEE, pp. 577–582.

VirusTotal, 2020. File statistics. https://www.virustotal.com/en/statistics.

Zhou, C., Sun, C., Liu, Z., Lau, F., 2015. A c-lstm neural network for text classification. arXiv preprint arXiv:1511.08630.

**Weizhong Qiang** received the Ph.D. degree in computer engineering from Huazhong University of Science and Technology (HUST) in 2005. He is a professor at Huazhong University of Science and Technology, China. He is the author or co-author of about 30 scientific papers. His topics of research interests include system security about virtualization, cloud computing.

**Lin Yang** is currently a master student in cyberspace security at Huazhong University of Science and Technology (HUST), Wuhan, China. He received the B.S. degree in computer science and technology from the Northeastern University, Dalian, China, in 2018. His research interests include malware analysis and software security.

**Hai Jin** received the Ph.D. degree in computer engineering from the Huazhong University of Science and Technology (HUST), China, in 1994. In 1996, he was awarded a German Academic Exchange Service Fellowship to visit the Technical University of Chemnitz, Germany. He was with The University of Hong Kong from 1998 to 2000, and as a Visiting Scholar with the University of Southern California from 1999 to 2000. He is a Cheung Kung Scholars Chair Professor of computer science and engineering with HUST. He has co-authored 22 books and published over 700 research papers. His research interests include computer architecture, virtualization technology, cluster computing and cloud computing, peer-to-peer computing, network storage, and network security. He is a fellow of IEEE, a fellow of CCF, and a member of the ACM. He was awarded the Excellent Youth Award from the National Science Foundation of China in 2001.