Hindawi
Security and Communication Networks
Volume 2018, Article ID 5249190, 18 pages
https://doi.org/10.1155/2018/5249190

Research Article

LSTM-Based Hierarchical Denoising Network for Android Malware Detection

Jinpei Yan, Yong Qi, and Qifan Rao

Department of Computer Science and Technology, Xi'an Jiaotong University, Xi'an, Shaanxi, China

Correspondence should be addressed to Yong Qi; qiy@xjtu.edu.cn

Received 27 August 2017; Accepted 16 November 2017; Published 9 January 2018

Academic Editor: Tom Chen

Mobile security is an important issue on Android platform. Most malware detection methods based on machine learning models heavily rely on expert knowledge for manual feature engineering, which are still difficult to fully describe malwares. In this paper, we present LSTM-based hierarchical denoise network (HDN), a novel static Android malware detection method which uses LSTM to directly learn from the raw opcode sequences extracted from decompiled Android files. However, most opcode sequences are too long for LSTM to train due to the gradient vanishing problem. Hence, HDN uses a hierarchical structure, whose first-level LSTM parallelly computes on opcode subsequences (we called them method blocks) to learn the dense representations; then the second-level LSTM can learn and detect malware through method block sequences. Considering that malicious behavior only appears in partial sequence segments, HDN uses method block denoise module (MBDM) for data denoising by adaptive gradient scaling strategy based on loss cache. We evaluate and compare HDN with the latest mainstream researches on three datasets. The results show that HDN outperforms these Android malware detection methods,and it is able to capture longer sequence features and has better detection efficiency than N-gram-based malware detection which is similar to our method.

1. Introduction

Recently the rapid development of Android mobile system creates serious security problems. The quickly increasing number of mobile users promotes the emergence of a huge number of new Android applications, bringing users not only more application services but also potential malicious application threats. Manually detecting these malwares is impractical; the traditional malware detection method is based on signature, which generates a unique signature identifier for a malware. This signature is generated based on many attributes of malware such as file name, file content, or some manually extracted features. For an unknown program, the detection can be done by searching for a matching signature in a malware database. However, signatures only consist of a series of simple features; there is a high probability of escaping signature-based detection if malware is simply disguised by packing or obfuscating. And it also requires a large malware database continually updating to cope with new malwares emergence. So in recent years, malware detection based on machine learning is widely studied as a better alternative method.

At present, the existing malware detection methods are mainly divided into dynamic and static analysis. Dynamic analysis [1–3] can effectively resist packing and obfuscating operations, but requires to run real-time monitor applications, which are inefficient for taking up a lot of computation resources. Static analysis can be much faster and can be more suitable for detecting a large number of applications. It includes permission-based approaches [4–6], N-gram-based statistical approach by mining byte or opcode sequences [7], high-level program analysis such as API call analysis, abstract syntax trees and control flow graphs analysis [8–10], and other well-designed malware analysis methods [11, 12]. However, these methods often heavily rely on expert knowledge to manually design features which are still commonly coarse-grained. The reason is that deep-level features are difficult to be discovered and designed, such as all kinds of suspicious opcode sequence patterns.

In this paper, we propose LSTM-based hierarchical denoise network (HDN) which directly learns from raw opcode sequences for malware detection. Since opcode sequences are often very long, HDN uses hierarchical structure to achieve effective modeling for very long sequences with LSTM, preventing the gradient vanishing problem. Concretely, HDN treats one opcode sequence as a composition of some subsequences which we call method blocks (the source code for Android application is made up of Java methods). Opcode encoder, as the first-level of HDN's hierarchical structure, learns a method block embedding by encoding it with LSTM, then method encoder as the second-level can learn and detect malware through method block sequence with another LSTM.

For a malware program all method blocks are regarded as malicious by default, but actually in some cases malicious code is implanted into a host application, in which the rest opcodes or what is called Java method blocks do not have malicious behaviors which should be regarded as noise during LSTM learning. To this end, we propose method block denoise module (MBDM) for HDN, using loss cache and adaptive gradient adjustment methods to denoise the input data when HDN is training. It reduces the defect of noisy method blocks on LSTM weight updating and filters out part of the noisy method blocks to achieve denoising. Overall, HDN can automatically learn features and patterns from raw opcode sequences to minimize the heavy requirement for expert knowledge. While compared with the similar work using N-gram-based malware detection, HDN can capture longer malicious opcode sequence patterns.

In the training process, we use a GPU to train HDN, where opcode encoder can run parallelly to compensate the low efficiency due to the serial computing limitation on LSTM. These greatly speed up the training of HDN. And once the training is completed, HDN can efficiently scan and detect an Android application through raw opcode sequences by a single forward propagation calculation, saving time on manual feature engineering. Hence, it is more effective compared with N-gram-based malware detection. Moreover, HDN can continually update network weights to adapt to the changing malware environment through incremental learning.

We evaluate HDN on three different datasets, which include two widely used malware datasets. In the first two benchmark datasets, HDN outperforms other related works on detection accuracy. In addition, we also make malware family classification experiment. Finally, in order to better simulate the real environment, the third dataset we use is a large "in the wild" dataset which contains plenty of latest Android samples. The results show that HDN is superior to N-gram-based malware detection both in detection accuracy and efficiency. In summary, we make the following contributions:

(i) We use LSTM, a deep neural network learning from raw opcode sequences to achieve malware detection, minimizing the heavy workload for artificial feature engineering.

(ii) We propose LSTM-based hierarchical denoise network using hierarchical structure to solve very long opcode sequence learning and gradient vanishing problems. Also, we propose MBDM for HDN to denoise opcode subsequences (method blocks) by loss cache and adaptive gradient adjustment, which helps HDN get better malware detection result.

(iii) We evaluate HDN with experiments on three different datasets. The result shows that HDN outperforms other related malware detection works. It greatly enhances the computation efficiency and detection accuracy comparing with the method based on N-grams.

The rest of the paper is organized as follows: Section 2 discusses the related work. Section 3 introduces our methodology and Section 4 presents the experiment and result analysis. Works are concluded in Section 5.

2. Related Work

The detection and analysis of malware have always been a research area of concern. In this section, we will review the current malware detection technologies and then introduce relevant detailed works for very long sequences learning (since our model needs to deal with very long opcode sequence data).

2.1. Android Malware Detection. Malware detection methods can generally be divided into dynamic analysis and static analysis, and most of current static analysis methods are based on machine learning technology [5, 6, 8, 9, 13–16] which is gradually replacing the traditional signature-based methods. Concretely, Barrera et al. [5] first proposed to use a permission-based approach for malware detection. Peng et al. [6] proposed an improvement approach by applying probabilistic generation model to permissions. Aafer et al. [9] proposed a method called DriodAPIMiner, which uses kNN classifier to analyse features extracted from API level. Arp et al. [13] added some interpretation of the detection result as an improvement; they proposed a light-weight malware detection method called Drebin, which uses static features such as application permission, API calls, and network address combined with SVM to detect malwares. Similarly, Cen et al. [15] extracted import information (API calls, class information, and other features) and used the probability discrimination model to achieve malware detection. Recently, there is an idea using ensemble learning to blend API calls and user intentions, permissions, and code commands for malware detection [16].

Chen et al. [8] came up with a novel work which compared the UI common points to find malware repackaged from the original and then associated different Android views and control flow graph features through user interaction codes to find the differences and which one is malicious. Rastogi et al. [17] tried to explore the effects of packing and obfuscating transformation attacks and suggested that malware detection should rely on semantic analysis rather than statistics on API call or code strings since the former

is not susceptible to transformation. They also disagreed with analysing a high-level source because it can be easily obfuscated. There are other well-designed malware analysis works, like Du et al. [18] who used D-S evidence theory with feature extraction (permissions, APIs, and control flow graph) to do malware detection. Xu et al. [12] discovered that some malware programs do not apply for their sensitive permissions directly, but used intercomponent communication (ICC) to call other components to achieve sensitive operations.

Instead of the above use of the advanced manual-designed features, we extract the sequence features from the raw opcode sequence of a program, which is inspired by related works [19, 20] on Windows OS platform. The most similar work to what we do is using an N-gram model on opcode sequences for Android malware detection [7, 21]. Here N-gram is used to obtain the statistical characteristics from the opcode sequence. Jerome et al. [7] and Canfora et al. [21] show that just with 1-gram features, which are only the frequencies of each opcode, they can achieve a promising malware detection result. Also, the selection of parameter N for N-gram and the number of features extracted from raw N-gram features can greatly affect the accuracy of the malware classifier. However, just adding one for parameter N will result in a significant increase in computational consumption, which is the bottleneck for N-gram-based malware detection method. Besides, N-gram model usually requires extra feature selection to reduce the length of feature vector for computational efficiency.

In this work, we propose a method based on hierarchical LSTM network to automatically learn from raw opcode sequences for malware detection. It does not require manual feature engineering, while it is able to capture very long-range opcode sequence patterns and features.

2.2. Very Long Sequence Learning. LSTM has been proved to be an effective model learning feature from time series data. But in some time series data learning problem, the sequence length for a data sample is likely to be very long, far exceeding the length around 120 that LSTM is capable to learn, and we name it "very long sequence." In our scenario, the length of opcode sequence extracted from Android program is often very long. In particular, the average opcode sequence length reaches 36,000 in our dataset. Long sequences are difficult to be learned by LSTM. Although LSTM uses gate mechanism which allows the gradient backpropagated to the earlier time node to capture long-term timing dependencies and correlations compared with recurrent neural network. However, usually LSTM can only handle the length of the sequence within 200; otherwise, LSTM will consume a lot of time on error backpropagation calculation. While the calculation is inefficient due to the gradient vanishing or exploding problem, early time node can hardly get effective weight update.

There are several researches on processing very long sequences with LSTM. Pascanu et al. [22] used a simple method called "truncating and padding" to deal with very long sequence. It sets a fixed length N and truncates and discards the part of sequence exceeding N. And for the sequence of length less than N, it uses a predefined identifier to pad at end of the sequence to N. Here we can choose a unique element as the predefined identifier, and LSTM can automatically learn their "padding meaning" from data samples. However, an existing problem with this "truncating and padding" strategy is that if the fixed length N is too small, it will discard a large number of sequence information while if N is too large, it still cannot solve the gradient vanishing problem on LSTM.

Sak et al. [23] used truncated backpropagation through time (truncated BPTT) to train LSTM on very long sequences, where truncated BPTT [24] is a gradient calculation algorithm. It is similar to full BPTT which calculates gradient using BPTT algorithm over the entire sequence. Full BPTT is commonly used, but is ineffective if the sequence is too long. Considering BPTT algorithm cannot pass gradient to very early time point due to gradient vanishing, truncated BPTT uses a time window to limit the backpropagation distance. So BPTT gradient calculation is performed only inside the window, and the nodes outside the window do not get weight updating. Hence, this approach can achieve more efficiency by sacrificing a small part of accuracy.

In addition, truncated BPTT is also suitable for online learning where datasets are updating overtime. The most important thing for online learning is to quickly adapt to the newly generated data in time. If LSTM learns from input stream data which is constantly updated, then truncated BPTT will facilitate the processing of such data, which can be effectively updated as the input stream changes. Essentially, dynamically updated input streams can also be treated as a very long sequence.

Similarly, Doetsch et al. [25] proposed chunk BPTT by dividing long sequence into multiple chunks. Each chunk has the same length N, and the last chunk is padded to N. Then a number of chunks compose a minibatch for BPTT calculation. As a number of chunks can be calculated in parallel, it can achieve about 3 times computing acceleration for training very long sequences. Chen et al. [26, 27] proposed CSC-BPTT for improving chunk BPTT. Considering that the length of chunk is too short, the association between chunks cannot be learned and will affect the accuracy of LSTM model. Therefore, each chunk CSC-BPTT attaches the context of its adjacent two chunks to the beginning and the end, which eases the problem to a certain extent.

Li et al. [28] used an LSTM *seq2seq* autoencoder framework to form the representation of the document. The result indicates that the representation can preserve the semantic information of the document. It uses a hierarchical model first to obtain the representation of each sentence by an autoencoder then encodes the sequence of sentences to obtain the representation of the document. They use BLEU and other indicators to evaluate the reconstruction error between the reconstructed document and the source document. Yang et al. [29] then used a hierarchical model to classify documents by creating embedding for the word level and the sentence level, respectively.

In this paper, we propose a hierarchical structure to deal with long opcode sequences using LSTM, which is inspired by the work of Yang et al. [29]. In our scenario, we also

make two specific and important improvements. The first is modifying the error backpropagation calculation in our hierarchical structure. The original backpropagation error is transmitted only from the highest level down; in addition to that we calculate classification loss in the middle level of the hierarchical structure at the same time and then calculate gradient with BPTT. It accumulates the loss for the purpose of LSTM's weight updating more quickly, while avoiding the fact that some of the underlying weights cannot update effectively since the propagation distance is too far for the loss from top-level. Besides, hierarchical models are facing serious noise data interference when learning from raw opcode sequences, so we propose "method block denoise module" integrated in our hierarchy model during the training process to filter out noisy opcode subsequences, thereby improving both the detection accuracy and training efficiency.

3. Malware Detection Methodology

In this section, we introduce the overview of our malware detection method, explain how to get the opcode sequence from the Android application source file, and then describe how an LSTM-based HDN hierarchical model is designed and learned from raw opcode sequence to complete malware detection.

3.1. Overview. We now introduce the whole process of our Android malware detection method. First, we decompile the Android application source file (*.apk* file) by *Baksmali* to generate a *.smali* file. Then we extract raw opcodes from *.smali* file to form an opcode sequence; after that we transform the opcode sequence into vectors (we try two different methods to get vector representation, one is one-hot encoding method and the other is learning opcode embedding through data) and put these opcodes embedding into LSTM-based HDN to train a classifier for distinguishing malicious from benign samples. It should be noted that HDN has an important component called MBDM, which helps denoising the noise segments in the opcode sequence since not all opcode subsequences of a malware contain malicious behavior. We use MBDM to filter out the parts of opcode segments from a malware which do not contain malicious behavior for data cleaning. In particular, MBDM's denoising phase and HDN's training phase are designed to perform in parallel for efficiency. Finally, we get a classification result inferring an Android application as either malware or benignware by LSTM-based HDN. The overview of our detection process is depicted in Figure 1 and the rest of this section discusses each of these steps in detail.

3.2. Android Application Decompilation and Opcode Sequence Extraction. Android application is an *apk* package which contains *classes.dex*, *AndroidManifest.xml*, and other files. Here *classes.dex* file is executed on the Dalvik virtual machine (VM). Android app runs on specifically designed Dalvik VM by Google instead of the standard Java VM. The difference between the two is that Java VM uses Java bytecode while Dalvik VM uses Dalvik bytecode which is converted from Java bytecode and packaged into a Dalvik executable (DEX)

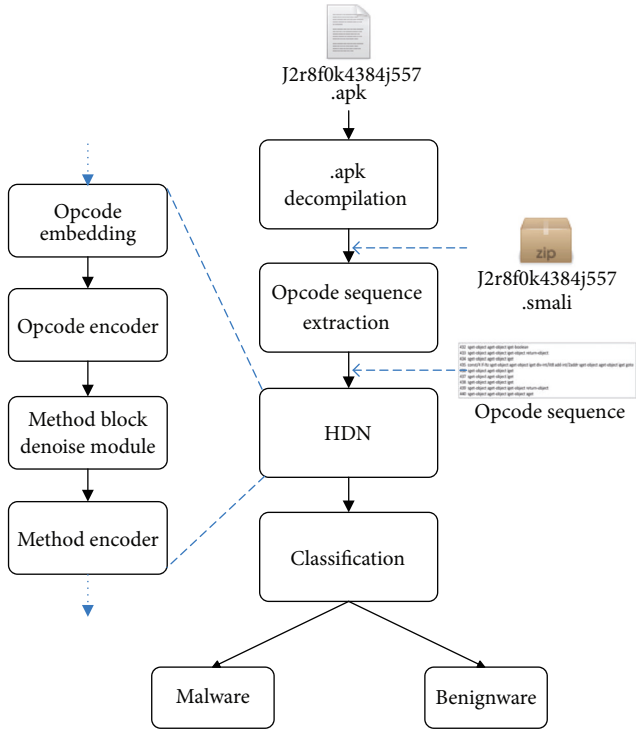

Figure 1: The overview of our malware detection process.

file. Android applications are compiled into *.dex* file, which can be created automatically by translating the compiled applications written in Java. So the *classes.dex* file contains Android source code information.

At present, the mainstream DEX disassemblers are *Dedexer* and *Baksmali*; here we use *Baksmali* to deal with *classes.dex* which generates *.smali* file as the decompiled result. The content of *.smali* file shows in Algorithm 1. Then we extract the opcodes to form an opcode sequence, collecting over 200 kinds of common opcodes, like mul-double&2addr, add-int, rsub-int, and so on. During this stage, we found that there are many consecutive repeating opcode subsequences in the raw opcode sequence. In order to reduce the length of the opcode sequence and increase the signal-to-noise ratio, we filter these repeating opcode subsequences to remove its redundant information (see in Figure 2).

3.3. Long-Short Term Memory (LSTM). Long-short term memory (LSTM) [30] is a powerful deep neural network for temporal data mining and learning, which is a variant of recurrent neural network (RNN). RNN uses recurrent connections within the hidden layer to create an internal state representing the previous input values, which allows RNN to capture temporal context. However, as the time interval expands, the updated gradient from BPTT would decay or explode exponentially, which makes RNN difficult to learn long-term features and dependencies. LSTM makes an improvement which takes a special module called constant error carousel (CEC) to propagate constant error signal through time, using a well-designed "gate" structure to

```
34 iget-object v0, v0, Lcom/admogo/AdMogoLayout;->custom:Lcom/admogo/obj/Custom;
35 move-object/from16 v34, v0
36 move-object/from16 v0, v34
37 iget-object v0, v0, Lcom/admogo/obj/Custom;->image:Landroid/graphics/drawable/Drawable;
38 move-object/from16 v34, v0
39 if-nez v34, :cond_53
40 invoke-virtual {v7}, Lcom/admogo/AdMogoLayout;->rotateThreadedNow()V
41 goto:goto_f
42 :cond_53
43 new-instance v9, Landroid/widget/ImageView;
44 invoke-direct {v9, v5}, Landroid/widget/ImageView;-><init>(Landroid/content/Context;)V
45 move-object v0, v7
46 iget-object v0, v0, Lcom/admogo/AdMogoLayout;->custom:Lcom/admogo/obj/Custom;
47 move-object/from16 v34, v0
48 move-object/from16 v0, v34
49 iget-object v0, v0, Lcom/admogo/obj/Custom;->image:Landroid/graphics/drawable/Drawable;
50 move-object/from16 v34, v0
51 move-object v0, v9
52 move-object/from16 v1, v34
53 invoke-virtual {v0, v1}, Landroid/widget/ImageView;->setImageDrawable(Landroid/graphics/drawable/Drawable;)V
54 new-instance v11, Landroid/widget/RelativeLayout$LayoutParams;
```

Algorithm 1: The content of the decompiled *.smali* file generated by *Baksmali*.

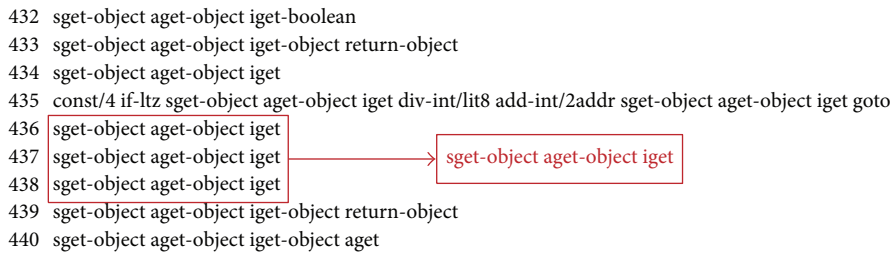

Figure 2: Consecutive repeating opcode subsequences in a raw opcode sequence.

prevent backpropagated errors from vanishing or exploding. The "gate" structure decides internal value of CEC according to current input values and previous context as it switches to control the information flow and memory. A standard gate contains a pointwise multiplication operation and a nonlinear transformation, so errors can flow backwards through a longer time range. There are three gates, named input gate, output gate, and forget gate, respectively.

Given an input sequence $X = \{x_1, x_2, \ldots, x_T\}$, where input gate, forget gate, and output gate in LSTM structure, respectively, are notated as i_t, f_t, and o_t and the weights attached to them are W_i, W_f, W_o, b_i, b_f, b_o. For each time step, LSTM updates two states, hidden state h_t and cell state c_t, and σ denotes the sigmoid function. The above parameters are presented as follows:

$$
\begin{aligned}
f_t &= \sigma\left(W_f * \left[h_{t-1}, x_t\right] + b_f\right),\\
i_t &= \sigma\left(W_i * \left[h_{t-1}, x_t\right] + b_i\right),\\
\widetilde{C_t} &= \tanh\left(W_c * \left[h_{t-1}, x_t\right] + b_c\right),\\
C_t &= f_t * C_{t-1} + i_t * \widetilde{C_t},\\
o_t &= \sigma\left(W_o * \left[h_{t-1}, x_t\right] + b_o\right),\\
h_t &= o_t * \tanh\left(C_t\right).
\end{aligned}
\tag{1}
$$

For a typical LSTM for classification, LSTM takes the normalized sequence data as input, and LSTM hidden layers are fully connected to the input layers. And there are recurrent connections for each LSTM hidden neuron. The size of LSTM output layer is equal to the number of categories to classify. But for a binary-class classification, a logistic regression is used for output layer to give a prediction between 0 and 1 at each time step and these predictions can be regarded as posterior probabilities of the input sequence belonging to the positive category at current time step.

3.4. Standard LSTM Architecture. We first build a standard LSTM as a base model, which learns from raw opcode sequences through a bidirectional LSTM. Bidirectional LSTM is almost the same as the LSTM structure

mentioned above, apart from the fact that it can take both past and future context into account, which is achieved by two separate hidden layers (forward and backward layer) dealing with past and future context, respectively, by propagating along sequence in opposite direction. And the output layer is connected to both hidden layers in order to combine past and future contexts. As mentioned above, for each Android application, we first decompile it by Baksmali and extract opcode sequences $X = \{x_1, x_2, \ldots, x_T\}$; these opcode x_t first need to be digitized as the input of LSTM. Here we use one-hot encoding to transform x_t into a sparse vector v_t (one-hot encoding simply uses a mapping transformation to get a sparse vector like $[0, 0, 0, 1, 0, \ldots, 0]$ which only contains one nonzero element) and use a bidirectional LSTM taking $\{v_1, v_2, \ldots, v_T\}$ as input. For ease of understanding, here we define LSTM (x_t, h_{t-1}) as the LSTM cell operation on the last hidden state h_{t-1} and the current input x_t

$$\begin{aligned} x_t &\xrightarrow{\text{One-hot}} v_t, \\ \overrightarrow{h_t} &= \overrightarrow{\text{LSTM}}\left(v_t, \overrightarrow{h_{t-1}}\right), \\ \overleftarrow{h_t} &= \overleftarrow{\text{LSTM}}\left(v_t, \overleftarrow{h_{t-1}}\right). \end{aligned} \tag{2}$$

For a malware detection task, it is a binary-class classification. We assume that the positive class label (malware) is 1 and the negative class label (benignware) is 0. The output of LSTM is y, $y \in (0, 1)$, on behalf of LSTM determining the probability that current sample is positive. We use logistic regression to calculate the probabilities (see (4)), where W_{LR} is logistic regression weight matrix, h is hidden state of LSTM for logistic regression, and b_{LR} is logistic bias. Since we use bidirectional LSTM, h contains the first and last two hidden states:

$$h = \left[\overleftarrow{h_1}, \overrightarrow{h_T}\right], \tag{3}$$

$$\begin{aligned} P(y = 1 \mid X) &= p(y \mid x_1, x_2, \ldots, x_T) \\ &= \frac{1}{1 + \exp\left(-W_{\text{LR}} h - b_{\text{LR}}\right)}. \end{aligned} \tag{4}$$

For multiclass classification tasks like malware family classification, the output of LSTM is y, $y \in \Re^{1*K}$, where K is the number of class. We use a softmax function to calculate the probabilities (see (5)); each class has its own parameter W_k, which is part of softmax weight matrix.

$$\begin{aligned} P(y_k \mid X) &= p(y_k \mid x_1, x_2, \ldots, x_T) \\ &= \frac{\exp\left(W_k h + b_k\right)}{\sum_{k'=1}^{K} \exp\left(W_{k'} h + b_{k'}\right)}, \end{aligned} \tag{5}$$

where $b_{k'}$ is softmax bias.

During the training phase, each iteration calculates the log-likelihood loss for each sample according to the object function, and uses BPTT algorithm to update weights of standard LSTM, which is an extension of the backpropagation algorithm for temporal data. By adding a time dimension, the gradient can be passed through the timeline.

Since a very long sequence is difficult to be effectively process by LSTM, here we use "truncating and padding" strategy [22] to deal with the opcode sequence. First, it selects the fixed length of N and truncates the part exceeding length N, and then it uses a predefined identifier padding to length N if the opcode sequence length is less than N. This strategy ensures that the length of each opcode sequence that the LSTM needs to process remains consistent and not too long, while the drawback is that truncating operation causes a plenty of sequence information unused. The whole standard LSTM architecture is shown in Figure 3.

3.5. Hierarchical Denoise Network Architecture. As the core idea of detection process, we propose LSTM-based hierarchical denoise network to learn from raw opcode sequence. HDN consists of four parts: opcode embedding lookup layer, opcode encoder, method encoder, and MBDM. Actually, HDN acts as a binary classifier to solve malware detection problem. We mark malware samples with positive labels (+1) and benign samples with negative labels (0). The overall architecture is shown in Figure 4.

We first explain the notational conventions in this paper, P is defined as a program which may be malware or benignware. Since Android application is written in Java, whose source code can be seen as a composition of methods of different Java classes, we regard program P_i as a composition of a sequence of method blocks, $P_i = \{M_1, M_2, \ldots, M_L\}$. Each method block M_l contains a sequence of methods, $M_l = \{m_1, m_2, \ldots, m_s\}$, each method m_s is comprised of a sequence of opcodes, where $m_s = \{o_1, o_2, \ldots\}$, so M_l also can be represented as a sequence of opcodes, $M_l = \{o_1, o_2, \ldots, o_T\}$. We assume that each opcode o_t can be represented as a V-dimensional embedding vector e_t^o. Similarly, each method block M_l and program P_i can be represented as embedding vector e_l^{mb} and e_i^p through opcode encoder and method encoder, respectively. In particular, opcode encoder encodes a method block M_l with its embedding $\{e_1^o, e_2^o, \ldots, e_T^o\}$ by LSTM to get method block embedding e_l^{mb}, then method encoder deals with method block sequence $\{M_1, M_2, \ldots, M_L\}$, encoding its embedding $\{e_1^{\text{mb}}, e_2^{\text{mb}}, \ldots, e_L^{\text{mb}}\}$ with LSTM to get program embedding e_i^p.

3.5.1. Opcode Embedding Lookup Layer. We try to use two methods to get opcode embedding, one is simply using one-hot encoding and the other is learning opcode embedding through training data. For the latter, an important thing is to set opcode vector's size. Since we have a total of 218 opcodes, the number of opcodes is not much comparing to the word embedding in natural language processing (NLP) task. So we choose embedding vector length V of 30 (refer to NLP task training *word2vec* word embedding, choosing embedding vector length of 300 for vocabulary with 20,000 words) and initialize an embedding lookup matrix W_e, $W_e \in \Re^{[218, V=30]}$. Before training, we use the uniform distribution of $[0, 1)$ to randomly initialize the embedding matrix and update it when the training loss propagates backward by BPTT to learn the

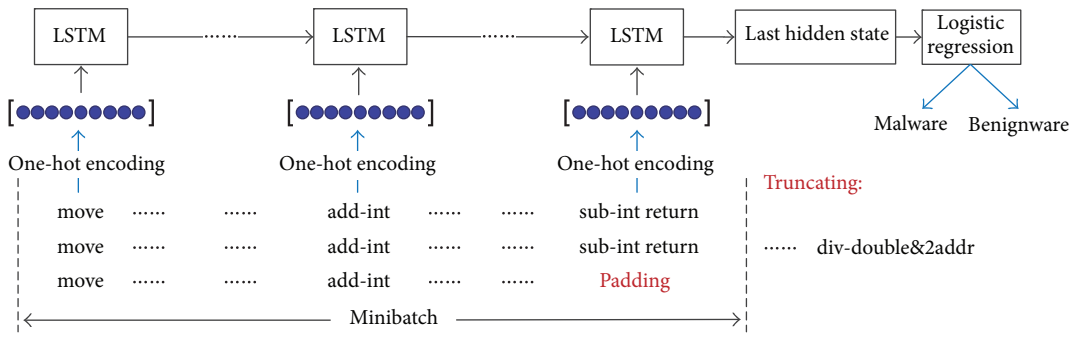

Figure 3: The whole standard LSTM architecture.

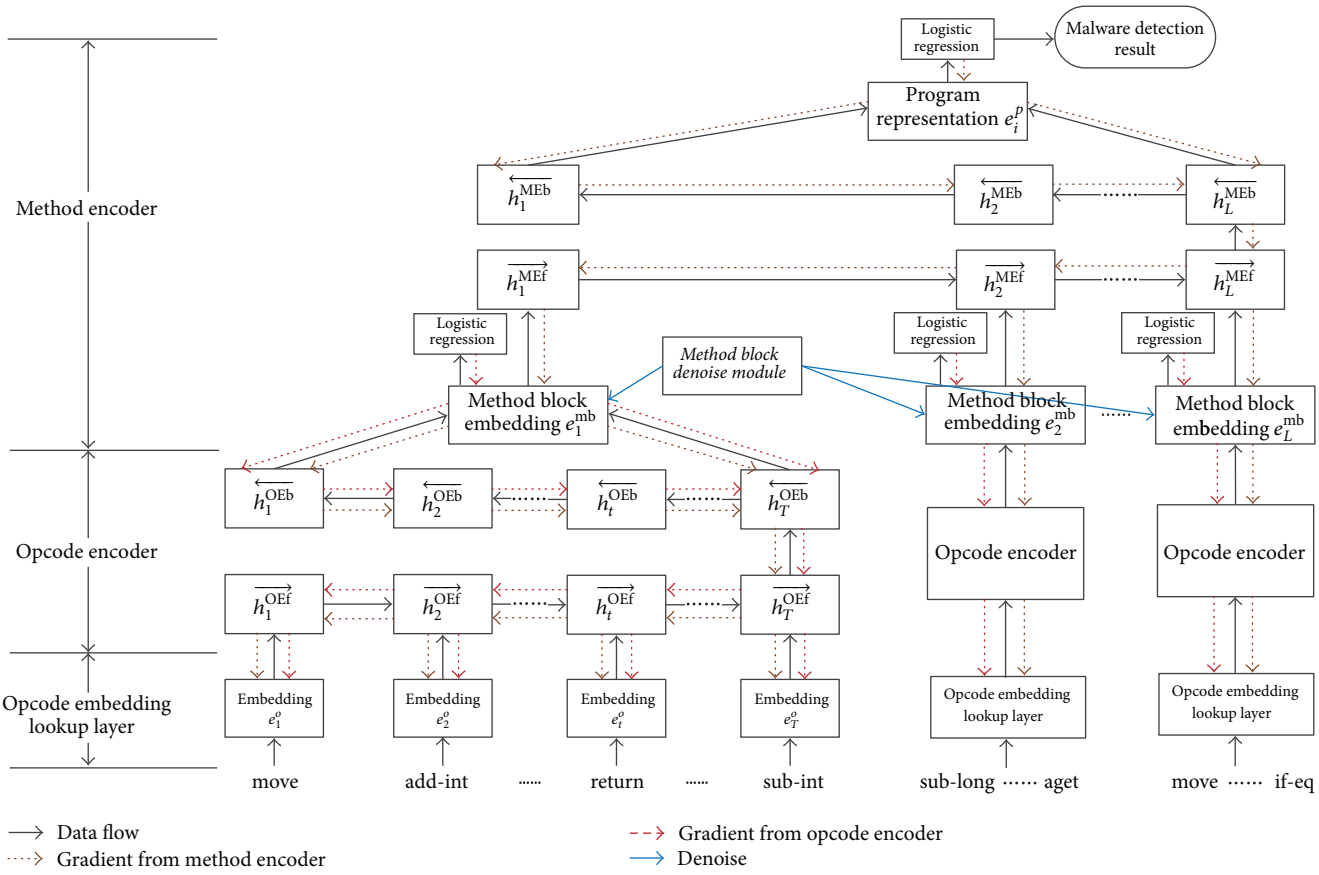

Figure 4: The overall architecture of HDN. Note that method block denoise module is used for all method blocks represented by blue line.

representation of the opcode from the data. For each opcode o_t, we get opcode embedding by looking it up and mapping it through W_e to digitize: $o_t \xrightarrow{W_e} e_t^o$.

3.5.2. LSTM-Based Opcode Encoder. As previously stated, we first decompile Android app to extract opcode sequence, because *Baksmali* tool will identify the scope of each Java method, using ".methods" and ".end methods" to identify during the decompiling process. We can recognize the opcode sequence boundary for a method, so the method is used as a basic unit for identifying opcode subsequence. In practice, as the lengths of Java methods are quite different, we define a method block M_l which may contain a long Java method or multiple short Java methods. So each of the method block has a similar opcode sequence length, which helps LSTM efficient training. Then we use opcode embedding sequences to represent method block $M_l = \{e_1^o, e_2^o, \ldots, e_T^o\}$.

Although we use method block to ease the varying opcode sequence length of different Java methods to a certain extent, method blocks still have different lengths. So we put

method blocks with similar lengths into a minibatch and pad them to the same length. It enhances the computational efficiency during LSTM training process, since the method block sequences only need to be padded to the longest sequence in the current minibatch rather than the longest one of the whole sequences.

We first digitize opcode with embedding matrix, then use opcode encoder's LSTM cell to encode opcode sequence of a method block like this:

$$\begin{aligned} o_t &\xrightarrow{W_e} e_t^o, \\ h_t^{\overrightarrow{\mathrm{OEf}}} &= \overrightarrow{\mathrm{LSTM}_{\mathrm{OE}}}\left(e_t^o, h_{t-1}^{\overrightarrow{\mathrm{OEf}}}\right), \\ h_t^{\overleftarrow{\mathrm{OEb}}} &= \overleftarrow{\mathrm{LSTM}_{\mathrm{OE}}}\left(e_t^o, h_{t-1}^{\overleftarrow{\mathrm{OEb}}}\right), \end{aligned} \tag{6}$$

where $\overrightarrow{\mathrm{LSTM}_{\mathrm{OE}}}$ represents LSTM cell in opcode encoder. $h_t^{\overrightarrow{\mathrm{OEf}}}$ and $h_t^{\overleftarrow{\mathrm{OEb}}}$ represent the opcode encoder's forward and backward hidden states, and t refers to the time step. Here we collect the first and the last hidden states to form the method block embedding $e_l^{\mathrm{mb}} = [h_1^{\overleftarrow{\mathrm{OEb}}}, h_T^{\overrightarrow{\mathrm{OEf}}}]$.

Unlike other related work [29], here we also calculate a logistic loss for opcode encoder. The intuition behind it is that if the logistic loss is calculated only from the highest level of the program, which needs to be passed from the method encoder through opcode encoder, and then to the opcode lookup layer, it may make the underlying weight hard to be effectively updated due to the too long distance, so we calculate logistic loss for both the opcode encoder and method encoder, where the loss of the method encoder is the "program classification loss". Similarly, we mark each method block with the same label as the program sample it belongs to. So the output of opcode encoder can also be classified so to get "method block classification loss," as shown as follows:

$$\begin{aligned} P\left(y_l = 1 \mid M_l\right) &= p\left(y_l = 1 \mid o_1, o_2, \ldots, o_T\right) \\ &= \frac{1}{\exp\left(-W_{\mathrm{LR}}^{\mathrm{mb}} e_l^{\mathrm{mb}} - b_{\mathrm{LR}}^{\mathrm{mb}}\right)}, \\ L^{\mathrm{mb}} &= -\left[\sum_{l=1}^{L} y_l \log\left(p\left(y_l = 1 \mid M_l\right)\right)\right. \\ &\quad \left. + \left(1 - y_l\right) \log\left(1 - p\left(y_l = 1 \mid M_l\right)\right)\right], \end{aligned} \tag{7}$$

where y_l is the method block's true label, $y_l \in \{0, 1\}$. $W_{\mathrm{LR}}^{\mathrm{mb}}$ is logistic weights matrix for method block. $b_{\mathrm{LR}}^{\mathrm{mb}}$ is logistic bias for method block.

3.5.3. LSTM-Based Method Encoder. From above, we get each method block M_l's embedding e_l^{mb} through the opcode encoder. An Android program is composed of a series of method blocks, so we use method encoder as the second-level encoder, and encode method block sequence $P_i = \{M_1, M_2, \ldots, M_L\} = \{e_1^{\mathrm{mb}}, e_2^{\mathrm{mb}}, \ldots, e_L^{\mathrm{mb}}\}$ to get the program embedding e_i^p. e_i^p as a high-level representation of the program, then it is used as features for the subsequent malware detection. The prediction result is generated by logistic regression layer, and HDN's weights are updated by BPTT algorithm, as shown below:

$$\begin{aligned} h_l^{\overrightarrow{\mathrm{MEf}}} &= \overrightarrow{\mathrm{LSTM}_{\mathrm{ME}}}\left(e_l^{\mathrm{mb}}, h_{l-1}^{\overrightarrow{\mathrm{MEf}}}\right), \\ h_l^{\overleftarrow{\mathrm{MEb}}} &= \overleftarrow{\mathrm{LSTM}_{\mathrm{ME}}}\left(e_l^{\mathrm{mb}}, h_{l-1}^{\overleftarrow{\mathrm{MEb}}}\right), \\ e_i^p &= \left[h_1^{\overleftarrow{\mathrm{MEb}}}, h_L^{\overrightarrow{\mathrm{MEf}}}\right], \\ P\left(y_i = 1 \mid P_i\right) &= p\left(y_i = 1 \mid M_1, M_2, \ldots, M_L\right) \\ &= \frac{1}{\exp\left(-W_{\mathrm{LR}}^p e_i^p - b_{\mathrm{LR}}^p\right)}, \\ L^p &= -\left[\sum_{i=1}^{I} y_i \log\left(p\left(y_i = 1 \mid P_i\right)\right)\right. \\ &\quad \left. + \left(1 - y_i\right) \log\left(1 - p\left(y_i = 1 \mid P_i\right)\right)\right], \end{aligned} \tag{8}$$

where $\overrightarrow{\mathrm{LSTM}_{\mathrm{ME}}}$ represents LSTM cell on method encoder. $h_t^{\overrightarrow{\mathrm{MEf}}}$ and $h_t^{\overleftarrow{\mathrm{MEb}}}$ represent the method encoder's bidirection hidden states. W_{LR}^p is logistic regression weights matrix, and b_{LR}^p is logistic regression bias. Similarly, we collect the first and the last hidden states to form the program embedding e_i^p.

In summary, HDN mainly uses opcode encoder and method encoder to form a two-level hierarchical model. Respectively, they get method block embedding e_l^{mb} and program embedding e_i^p and calculate "program classification loss" L^{mb} and "method block classification loss" L^p for HDN network learning and weight updating.

3.5.4. Method Block Denoise Module. Android malware is often deposited on a host application, so only part of the opcode sequence contains malicious characteristics or behaviors. However, HDN considers all opcode segments of a malicious sample malicious by default. In this situation, the benign (or not malicious) segments of a malware bring in the noise which reduces the quality of malware data samples, thus, requiring a denoising method. Otherwise, it will interfere with HDN's learning process to a certain extent. Here, we use MBDM to denoise during HDN training phase (see in Figure 5).

Our method is inspired by a work which uses reconstruction errors to detect anomaly sequences with LSTM Autoencoder. In our scenario, we use logistic loss as a reference to denoise subsequence (method blocks). For each opcode sequence, LSTM typically iterates it over the minibatch many times for training. MBDM sets a loss cache ϵ_j for each method block and accumulates logistic loss of method encoder for current method block, each iteration as followed:

$$\epsilon_j = \epsilon_{j-1} + \alpha_j * L_j^{\mathrm{mb}}, \tag{9}$$

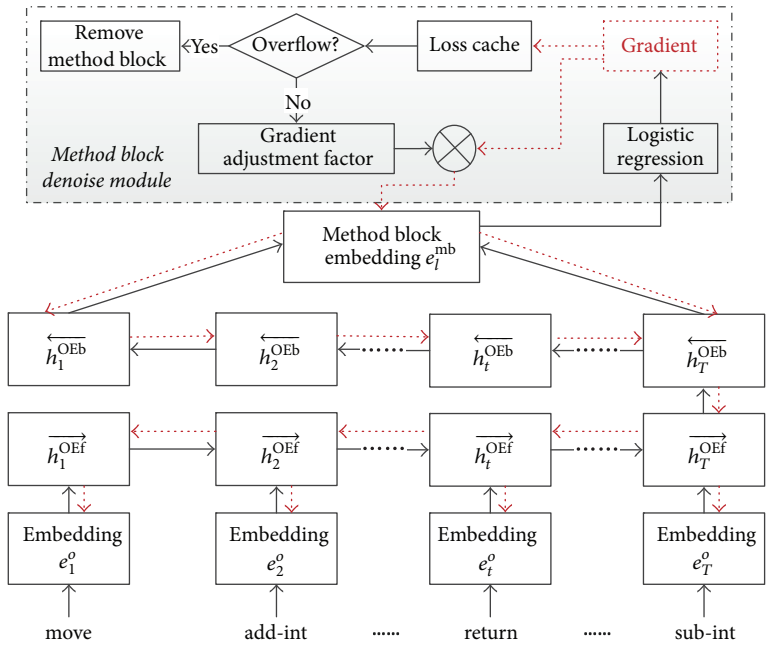

FIGURE 5: The flowchart of MBDM.

where j represents the current iterations round for current method block. L_j^{mb} represents "method block classification loss" in this iteration. α_j is the weight of current logistic loss.

During the initial stage of training, since the weights on method encoder are randomly initialized and HDN has not learned enough features from data, all method blocks' logistic loss are large so it is difficult to judge whether the method block is noisy. Hence, we adjust α_j to reduce the accumulative error at the beginning like this:

$$\alpha_j = \begin{cases} \alpha_0 + \log \frac{j}{\beta}, & L_j^{mb} > L_{j-1}^{mb}, \\ 0, & \text{otherwise}, \end{cases} \tag{10}$$

where α_0 and β are hyperparameters to be chose.

Then MBDM adaptively adjusts the gradient for back-propagation according to loss cache ϵ_t. In particular, the loss cache is mapped to for changing the amplitude of gradient as follows:

$$W_j = W_{j-1} - \eta * \left\{ \frac{1}{1+\epsilon_j} \nabla E\left(W_{j-1}\right) \right\}. \tag{11}$$

Intuitively, the noise subsequence is difficult to classify by LSTM due to the lack of effective information. Therefore, the logistic loss and the loss cache are relatively larger. We clip the gradient of these subsequences to avoid the noise subsequences giving the wrong weight updates to method encoder. Furthermore, if the loss cache exceeds the preset upper bound δ, this method block will be removed from the current program sample to avoid invalid gradient calculation.

3.5.5. Data Augmentation for HDN. In practice, the number of each class from the dataset is often very uneven. For example, the number of benignware is much more than that of malware and is easy to obtain. While the distribution of different Android malware families (a malware family refers to a malware variants group with homogeneous attack behaviors) is very uneven, malware families which are widely spread have a bigger influence on training model. Directly using these datasets which contain obvious class imbalance will seriously decrease the model's training performance. A common solution is to use data augmentation strategy to rebalance the distribution. One typical example is to do a certain mapping transformation to expand the image samples in computer vision tasks.

For time series data, we propose a novel data augmentation method for HDN (see in Figure 6). The HDN's first level is opcode encoder dealing with method blocks, and each method block M_l contains one or some methods, here we introduce some randomness when allocating methods to method blocks. Let each method block M_l contain $1 - N$ random methods m_s, and method block windows allow intersection. Given a program's opcode sequence, there are a number of method block allocation ways. So HDN can use all these different allocations as expanded new samples.

4. Experiments and Evaluations

After presenting HDN in detail, we now evaluate its performance. In particular, we conduct the following experiments. First, we describe the experiment environment and implementation process of HDN. Second, we evaluate and analyse the detection performance of HDN. Finally, we present the visualization for opcode embedding learned by HDN.

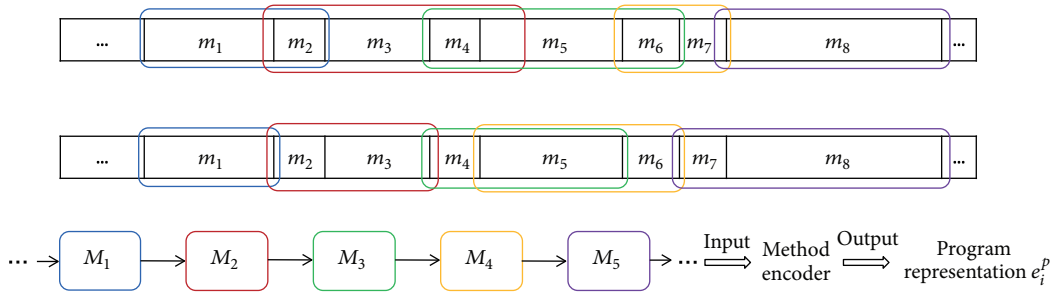

Figure 6: The data augmentation method for HDN.

Table 1: The datasets used in our work.

Datasets	Malware			Benignware	
	Source	samples	Malware families	Source	Samples
Benchmark dataset 1 (BD1)	Genome	1,260	49	China Android markets	1,250
Benchmark dataset 2 (BD2)	Drebin	5,560	179	China Android markets	5,600
In the wild dataset (ITW)	Androzoo	20,000	208	Androzoo	20,000

4.1. Implementation

4.1.1. Datasets. We use three malware datasets for our experiment, namely, Genome [31], Drebin [13], and Androzoo [32]. The first two datasets provide malicious Android app samples, which are being kind of out-of-date by now. However, they are widely used as benchmark datasets for many related works, so here we use these two datasets as a benchmark to compare our method with others. At the same time, we collect the latest Android malware and benign samples from Androzoo, which is a growing dataset for Android applications collected from several sources, like the official Google Play app market, and each of them has been analysed by tens of different antivirus products to detect and check which one is a malware. Our collected samples include most recently 20,000 benign samples and 20,000 malware samples, respectively. The malware samples contain 208 malware families, with an average of 100 samples per malware family.

In addition, we crawl on mainstream China Android markets (Wandoujia, Android markets, An Zhi, JiFeng, etc.) to get a number of the benign datasets. All these benign samples are validated by VirusTotal [33] to prevent wrong labeling. Overall, we finally build three complete datasets (containing both malware and benignware) which are described in Table 1.

In Table 1, each sample carries with the *.apk* a source app file in all datasets. The first malware dataset is Genome (called the Android Malware Genome project), and these samples are collected in 2012. In our work, we use Genome to build the malware part of our first benchmark dataset BD1. The second malware dataset is Drebin, which is the largest public dataset available before 2014. This dataset provider also came up with a good malware detection method and evaluated it on their dataset in comparison. We use Drebin to build the malware part of our second benchmark dataset BD2. The third malware dataset source is Androzoo. It is a large, constantly updated Android malware database compared with the previous two datasets, containing many samples collected after 2015. To ensure the balance of the number of malware family samples, here the number of samples from each malware family we collect is around 100, and we call it "in the wild dataset" ITW.

The goal of our malware detection experiment is to determine whether data samples belong to malware, which can be regarded as a binary classification problem. To ensure the reliability of results and make full use of data samples, we use 8-fold cross validation for evaluation and do an average as the final result. Certainly, to ensure that training samples and test samples have similar malicious/benign proportions, the data samples are required to shuffle at first.

Moreover, to quantify and evaluate classification results, we measure six indexes: accuracy, true positive rate (TPR), false positive rate (FPR), equal error rate (EER), receiver operating characteristic (ROC), and training/detecting time consumption, where TPR indicates the rate that a malware sample is correctly identified and FPR indicates the rate that a benignware sample is wrongly identified as a malware. In addition, since some papers use precision, recall, and F-score indicators for evaluation, we also calculate these indexes to facilitate the comparison.

4.1.2. Model. In our scenario, we use NVIDIA GTX980 GPU, which provides a high-performance CUDA universal parallel computing platform to support GPU computing and introduce GPU-accelerated deep neural network library cuDNN based on CUDA. According to the corresponding version of GTX980, the experiment platform mainly uses CUDA 7.5, cuDNN V4, Python 2.7.6, Numpy 1.8.2, Scipy 0.13.3, and Tensorflow 0.9.0. In particular, Tensorflow is an open source library mainly for deep learning tasks developed by Google. It helps researchers and engineers build and train flexible and customizable deep neural networks.

First, we introduce the specific settings of model and the related parameters for HDN. Our model uses bidirectional LSTM unit with dropout strategy (dropout probability is chosen to 0.5) to reduce overfitting problem, and dropout only takes effect on LSTM hidden layers. We use orthogonal initialization strategy to increase convergence of LSTM. The initial weight range for the network is $[-0.04, 0.04]$. The number of hidden layers is one and the number of hidden nodes is 650 for both opcode encoder and method encoder, with *tanh* as the activation function.

HDN uses Adam optimization algorithm for training, which is an optimization algorithm able to adaptively adjust learning rate during training phase, only requiring to set the initial learning rate which we set to $2e-3$. HDN uses minibatch training with stochastic gradient descent (SGD), where the batch size is selected as 30. Each method block length is between 80 and 160. For some method length more than 160, they will be divided into multiple method blocks for training. The maximum number of iterations is 160,000. Since using early stop strategy to prevent overfitting, each training sample iterates around 7 times on average.

We try two opcode embedding methods, one simple way is using one-hot encoding for representation and the other is using HDN learning opcode condense embedding from the data to build an embedding matrix. It should be noted that the learning rate for the embedding matrix should be much less than HDN network's learning rate, otherwise, it will lead to serious overfitting. Here the learning rate we set for the embedding matrix is 1/10 of the HDN network.

4.2. Results and Analysis

4.2.1. HDN-Related Experiment Results. Here, we make some experiments to explore the performance of HDN compared with standard LSTM model, we also evaluate the effect of MBDM and two opcode encoding methods to better optimize HDN model.

First, we try to prove that HDN is better to deal with long sequences than standard LSTM due to its hierarchical structure. Here standard LSTM uses truncation and padding strategy to handle long raw opcode sequences, which requires to set a fixed length N as a hyperparameter. It will truncate the part of sequence exceeding length N or padding to N if opcode sequence length is less than N. Since the selection of N greatly influences the results of standard LSTM, we try a set of different N for experiment. The results are shown in Figure 7.

It can be seen that HDN's detection AUC is 0.99848, which is a significant improvement compared with standard LSTM. One main reason is that standard LSTM truncates the opcode sequence for the sake of computational efficiency, which makes the information of the truncated part unutilized, leading to a great negative impact on standard LSTM's learning process. In contrast, HDN copes with the vast majority of long sequences very well, and its hierarchical structure allows HDN to handle a much longer sequence than standard LSTM for the sequence length that each layer LSTM in HDN needs to process is in a reasonable range. HDN uses a strategy similar to divide and conquer algorithm, which breaks opcode sequence into subsequences, then it deals with opcode subsequence segments (method blocks) by opcode encoder, getting higher and denser abstract representation, and then it unifies them for global processing by method encoder.

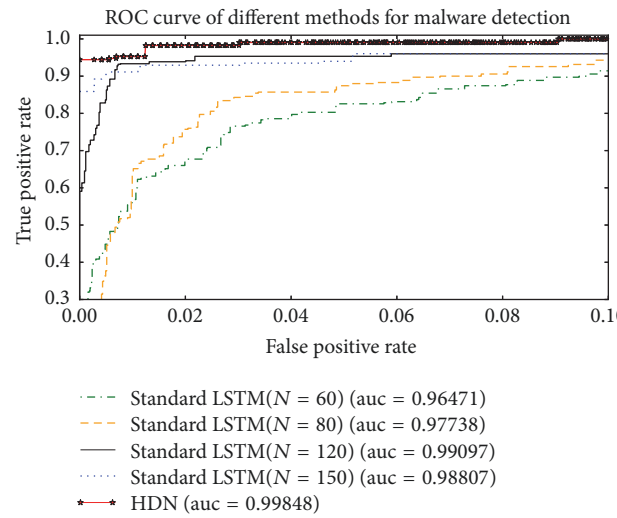

Figure 7: The performance comparison for different malware detection methods.

Moreover, we compare HDN with or without MBDM to verify whether MBDM can effectively remove the noise in sequence and enhance overall detection performance. Since MBDM involves some hyperparameters, we do not specifically optimize these hyperparameters in this phase. Here we simply choose a set of hyperparameters based on experience. The experiment results are shown in Table 2.

As can be seen from the results, MBDM brings a considerable improvement on detection rate for HDN proving that using MBDM to denoise opcode sequences is effective. Moreover, from the experiment we observe that MBDM can speed up the training process of HDN, because MBDM filters some noise subsequence segments so that it reduces the number of segments that HDN required to process. In general, it enhances the efficiency of learning and reduces time consumption of training.

We also compare two opcode encoding methods. One simply uses one-hot encoding. Since the number of Android opcode is not very large, the sparseness of one-hot encoding does not cause too much negative impact on computational efficiency. Another method is learning opcode embedding from data samples. First, we set the size of embedding vector with hyperparameter *emb_size,* randomly initialize the embedding lookup matrix with size of [*emb_size, num_opcode*], and then update the weight of the embedding lookup matrix to learn the dense vector representation of opcode during training phase. It's just like updating the weight of HDN network through BPTT. The idea first comes up in NLP tasks for word vector learning, such as well-known *word2vec*. Many previous works show that *word2vec* has an obvious good effect on NLP tasks, for example machine translation and machine reading comprehension. It should

Table 2: The malware detection results of different methods.

Methods	Accuracy (%)	AUC	TPR (FPR = 1%)	EER (%)
Standard LSTM (N = 60)	93.65	0.968	54.31	5.78
Standard LSTM (N = 80)	94.13	0.977	55.02	4.92
Standard LSTM (N = 120)	96.85	0.991	92.91	4.11
Standard LSTM (N = 160)	96.69	0.991	90.91	4.27
HDN (no MBDM)	98.86	0.997	96.69	1.39
HDN + one-hot encoding	99.13	0.998	97.73	0.98
HDN + opcode embedding	*99.42*	*0.999*	*98.88*	*0.73*

be noted that the learning rate for embedding lookup matrix should be much smaller. Here we scale to 1/10, otherwise, it will cause serious overfitting problem.

As summarized in Table 2, using opcode embedding achieves the better detection results, since opcode embedding captures the opcode semantic information compared to one-hot encoding, which helps HDN learn semantic information of opcode sequences and automatically mine malicious behaviors. In the last evaluation section, we will visualize the opcode embedding to show how semantic knowledge is learned and represented.

Besides, in order to bring deeper intuition about how opcode sequence length impacts the detection performance for the methods mentioned above, we experiment with several malware detection tasks using opcode sequence data with different lengths. Since the size of Android *.apk* samples vary a lot in our datasets, we collect the samples of similar opcode sequence lengths to generate one subdataset, which constructs a total of 9 subdatasets with an opcode sequence length ranging from 40 to 2,048, respectively. For each subdataset, we use three methods (standard LSTM, HDN, and HDN (no MBDM)) to experiment on all the subdatasets separately. The results are shown in Figure 8.

As we can see from Figure 8, all three methods' performance is basically the same in the first four subdatasets (which opcode sequence length is in the range of 40–120). But when opcode sequence length is greater than 120, the performance of standard LSTM dropped drastically due to the fact that standard LSTM is unable to effectively deal with long time series data. In contrast, HDN uses a hierarchical structure so even when the opcode sequence length exceeds 1,000, it is still possible to efficiently capture sequence features. And the use of MBDM brings an obvious improvement when opcode sequence length is greater than 1,000. Because the large Android malware is often generated by injecting malicious fragments to a benign application which contains a large number of nonmalicious noise, using MBDM can obtain a better detection performance when dealing with very long sequence due to the effectiveness of noise filtering.

From the above experiments we know that MBDM helps HDN get a better detection result through subsequence denoising, we intend to further explore and optimize the hyperparameters for MBDM. The goal of MBDM is to filter out the "noise subsequence" with low signal-to-noise ratio so that more quality input data is received by HDN. In our scenario, noise subsequences indicate noise method blocks,

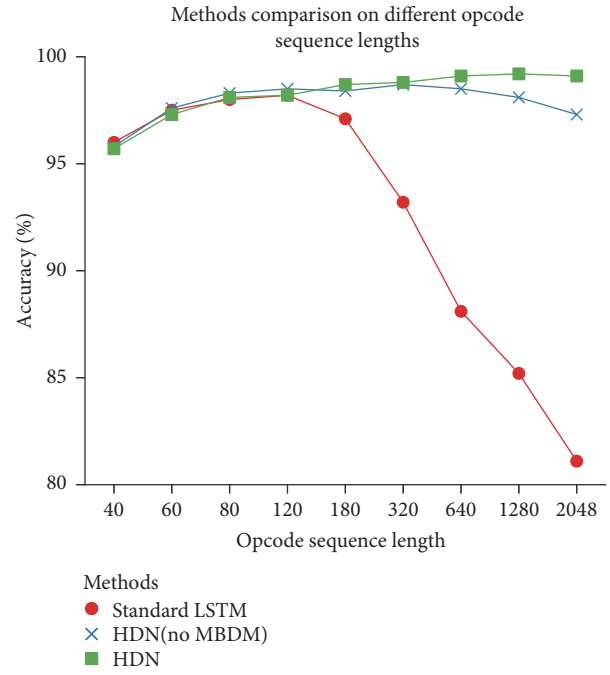

Figure 8: The methods comparison on different opcode sequence lengths.

so we will filter out the current method block once its loss cache overflows the upper limit. Here this requires a reasonable value for the upper limit of the loss cache, and since this upper limit is too abstract, we transform it into another hyperparameter called filtering ratio σ whose value refers to the proportion of training subsequences filtered out by MBDM in the training process. The choice for σ is related to the degree of tolerance for noise in opcode sequences, and in fact it reflects the trade-off between HDN generalization performance and data quality.

In this paper, we evaluate five different filtering ratio ratios σ (= 0%, 1%, 2.5%, 5%, 10%), where σ = 0% means that MBDM is not enabled. The results are shown in Figure 9. It can be seen that the result of σ = 10% corresponds to the lowest AUC value, and the rest of the filtering ratio σ get better results than the one without MBDM. From the experiment result the optimal filtering ratio σ is chosen to be 2.5%.

In order to explore the reason why the accuracy is declining when σ > 5%, a potential factor is that LSTM

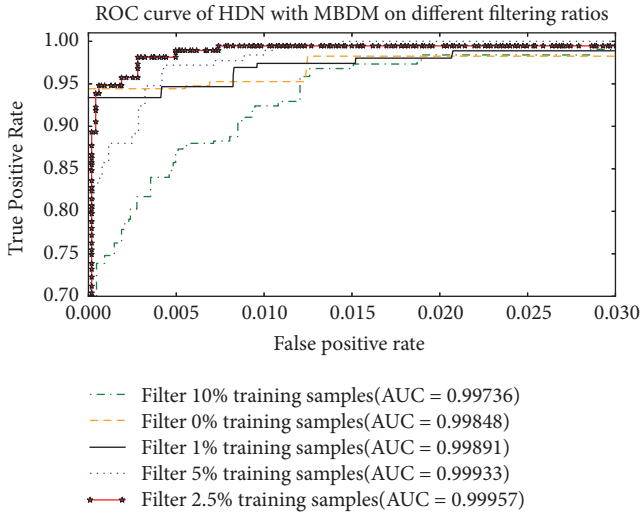

Figure 9: The comparison for different filtering ratios σ for MBDM.

network has strong generalization ability, that is, the input data with certain level noise does not affect LSTM to do classification much. LSTM can learn to ignore the noise from the data after several iterations. In the field of computer vision, in order to enhance the generalization performance of deep neural networks, researchers even introduce additional noise into the input image, such as masking parts of the image deliberately. As mentioned above, choosing the filtering ratio for MBDM requires a trade-off between data quality and model generalization performance. Although filtering more data samples can eliminate more data noise, the data information that HDN can learn is also relatively less. As we can see when $\sigma = 10\%$, HDN cannot learn a good weighted network since the valid data is not enough after MBDM filters.

4.2.2. Comparison with Other Related Works. We compare HDN with other mainstream methods on BD1 and BD2 benchmark datasets. Since different statistical indicators used in different papers vary a lot, we calculate all the various indicators to facilitate the comparison. The experiment results are shown in Table 3.

Comparatively, it can be seen that HDN has obtained a better detection result on both datasets. On the BD1 dataset, the contrast method includes using API dependency graph [34], feature extraction for raw opcode [7, 35], permissions-based method [36], API call frequency statistics [13, 37], and CFG-based method [38]. Although these related work results are considerable, they heavily require artificially features design on additional information and expert knowledge, such as permission request in *Androidmanifest.xml* or API calls information. In contrast, we only use raw opcode sequences to achieve malware detection, avoiding artificial feature extraction. The closest approach to HDN is Jerome et al. [7], who used N-gram model to extract features for raw opcode. The best result shown in their paper was achieved with nearly 200,000 5-gram features. It takes huge computation resources, as the computational complexity of N-gram increases exponentially with N. In the case of 5-gram ($N = 5$), a simple malware/benign pair extracts 1,305,511 different features, and they had to select 200,000 features with the most information gain through the feature selection method to reduce the computational cost.

The current computational efficiency requirements hardly satisfy the 6-gram and above, which actually limits the length of the extracted opcode sequence features, since the length of sequence dependencies beyond 5 will not be available captured by N-gram. However, due to the use of LSTM for sequence processing and hierarchical structure, HDN can capture more long-term sequence features compared with N-gram. It is obvious that HDN enhances the detection result compared with N-gram model.

On the BD2 dataset, we also compare with a novel method based on ICC features called ICCDetector [12]. Due to the fine-grained analysis on raw opcode sequence, HDN outperforms methods based on artificial features extraction (e.g., ICC, APIs, permissions). We can see that the closest detection result to HDN is ICCDetector. They found that some malware programs do not directly apply for sensitive permissions, but using ICC to call other components to achieve sensitive operations, so they extracted some of ICC-based features to achieve malware detection. However, this is not a common feature for all malware programs, so it has to combine with permission-based features when facing malware directly applying for sensitive permissions. These feature combinations require experience and a lot of attempts. From another point of view, ICC-related method is one specific opcode sequence call in assembly perspective, so HDN may even automatically mine ICC-call related opcode segment features from massive raw opcode sequences.

In addition, we do experiments in a large, real-world dataset called ITW dataset. This dataset is collected by us and no related work has done benchmark test for comparison. Here we reproduce the N-gram-based method according to related work. Considering calculation efficiency we conduct two 2-gram and 3-gram groups of experiments. Since 4 grams and above is difficult to meet the real-time calculation requirements on large training dataset, they are not taken into account. We first extract raw 3,132 2-gram features and 724,959 3-gram features, filter out a large number of low frequency features from raw 3-gram features followed by feature selection using random forest. The final features for each sample are constructed as a 12,834-dimensional feature vector. Also, since the choice of classifiers makes little effect on the results according to relevant research works, we use SVM as a classifier and do data preprocessing first to normalize the feature vector. The SVM classifier is built with LibSVM-3.2 tool developed by Professor Lin [41], which supports SVM both for binary or for multiple classification scenarios and provides automated scripts for hyperparameter tuning. The optimized hyperparameters in the experiment are chosen as $c = 1024$ and $g = 0.0097656$.

HDN's detection result on ITW large datasets is better than the previous two benchmark datasets, which is obviously better than N-gram results. Essentially, HDN is a deep learning method, which especially requires massive data for feature self-learning and modeling. Therefore, HDN can

Table 3: Comparing HDN with related works.

Methods	Features	Dataset	Benignware	Malware	Accuracy (%)	Precision	Recall	*F*1-score	TPR (%)	FPR (%)
DroidSIFT [34]	API dependency graph	BD1 (Genome)	13,500	1,260	-	-	-	-	98	5.15
CNN [35]	opcode patterns	BD1 (Genome)	863	1,260	98	0.99	0.95	0.97	-	-
DroidDetective [36]	permission-combination	BD1 (Genome)	741	1,260	96	0.89	0.96	0.92	-	-
Yerima et al. [37]	API calls, permissions, cmnds	BD1 (Genome)	1,000	1,000	91	0.94	0.91	0.92	-	-
Jerome et al. [7]	opcode *n*-grams	BD1 (Genome)	1,260	1,246	-	-	-	0.98	-	-
CSBD [38]	CFG	BD1 (Genome)	1,247	1,247	-	0.93	0.905	0.91	-	-
HDN	opcode patterns	BD1 (Genome)	1,260	1,260	*99.2*	*0.993*	*0.985*	*0.989*	*97.87*	*0.5*
Opcode ngrams [21]	opcode *n*-grams	BD2 (Drebin)	5,560	5,560	96.88	0.957	0.981	0.963	-	-
ICCDetector [12]	ICC-related feature	BD2 (Drebin)	12,026	5,264	97.4	-	-	-	93.1	0.67
RCP [39]	permissions	BD2 (Drebin)	123,453	5,560	-	-	-	-	17	1
KIRIN [40]	permissions	BD2 (Drebin)	123,453	5,560	-	-	-	-	39	5
Peng et al. [6]	permissions	BD2 (Drebin)	123,453	5,560	88.2	-	-	-	45	1
Drebin [13]	API calls, intents, permissions, cmnds	BD2 (Drebin)	123,453	5,560	-	-	-	-	93.9	1
HDN	opcode patterns	BD2 (Drebin)	5,600	5,560	*98.82*	*0.991*	0.977	*0.984*	*97.44*	*0.5*
2-gram SVM	opcode *n*-grams	ITW (Androzoo)	20,000	20,000	95.17	-	-	-	88.67	0.5
3-gram SVM	opcode *n*-grams	ITW (Androzoo)	20,000	20,000	96.14	-	-	-	90.02	0.5
HDN	opcode patterns	ITW (Androzoo)	20,000	20,000	*99.42*	-	-	-	*98.18*	0.5

Note. - means that the authors did not evaluate this index or did not mention it in the paper.

learn richer features and achieve better results in a large dataset. However, N-gram method is limited by the range of N and is not able to capture a higher dimension of sequence features, so it does not obviously benefit from the massive data.

4.2.3. Malware Family Classification. In addition to malware detection, we also complete the malware family classification experiment through HDN. The current mainstream antivirus tool used for software sample analysis not only determines whether it is malicious or not, but also shows which malware family it belongs to if meeting a malware. A malware family refers to malware variants group with homogeneous attack behaviors, and the classification of malware family helps build a high quality virus database and further analysis of malware attack mode.

The related work of malware family classification is relatively rare, but there are published results on BD1 and BD2's two datasets for malware family classification. So we use these datasets as performance comparison. The number of malware family samples in the BD1 and BD2 datasets is very uneven, and some malware family samples are too few to use the machine learning method to train. So we first make data preprocessing and count the number of all malware families for the BD1 dataset. The result shows training sample is unevenly distributed on malware families and the number is not sufficient (the largest number of malware families contains 300+ samples while the minimum only has 5 samples), the datasets are expanded with the data augmentation method based on the method blocks mentioned above. The effect of data augmentation is shown in Figure 10, where each malware family is augmented to around 800 samples, we adjust the augmentation ratio to maintain the approximate uniform distribution of different malware families to solve the uneven distribution of samples. Besides, we need to change the structure of HDN for multiclass malware family classification. The original output layer of HDN uses logistic regression for malware and benignware binary classification (malware detection), here the output layer is changed to softmax regression, with the rest of the structure and parameters staying the same.

Notice that TPR, FPR, and ROC curves are commonly used to evaluate binary classification results. To apply them on a multiclass classification problem, we use microaveraging method which regards all samples in each class as a two-class classification problem, that is, the sample of the target class is considered as positive and the sample of the other classes is considered as negative, averaging TPR and FPR values of all single class, then. The evaluation result is shown in Table 4.

HDN achieves a significant upgrade of classification accuracy compared with DroidSIFT on malware family classification task. DroidSIFT is based on the API dependency graph. The disadvantage of this approach is that decompilation is difficult to obtain API call information, since it belongs to high-level information and is susceptible to obfuscation operation, and modern malware uses more and more obfuscation and encryption technology to prevent from antivirus software tracking. In contrast, HDN learns from raw opcode sequences which are relatively low-level with raw data information, not easy to be obfuscated. Similar idea comes from DroidChameleon [17] that suggests semantics analysis for malware detection should rely more on bytecodes rather than source codes which are easy to be obfuscated or encrypted.

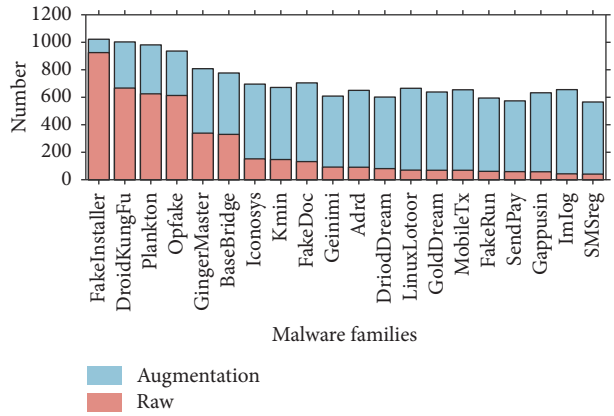

Figure 10: The results of data augmentation for malware family.

Moreover, HDN obtains TPR of 97.08% (FPR = 0.5%) on BD2 dataset for malware family classification, which is better both in TPR and FPR compared with Drebin [13]. The main reason is that Drebin uses manual-extracted statistical characteristics from permissions, single API calls frequency statistics, and network address information. It does not have a complete analysis for the malware program content. However, a malware program often initiates a malicious behavior call at the code level which contains rich information. In essence, HDN mining malicious behavior through opcode sequence is to capture such information.

4.2.4. Computation Efficiency. From the point of view of computational efficiency, we calculate the time consumption of different models in the training phase and the detecting phase, respectively (see in Table 5). It should be noted that the time consumption includes data preprocessing time such as feature extraction, but it does not include the time on decompiling the program due to its similarity for all models. We can see that HDN is superior to 3-gram SVM for computation efficiency in both training and detecting phases because the computational complexity of N-gram feature extraction increases exponentially with the size of N. When $N \geqslant 3$, the feature extraction takes a lot of time even if the training time of SVM is less than LSTM. And this feature extraction bottleneck is even more serious in the detecting phase.

On the contrary, deep neural network benefits from directly processing source samples and achieves considerable detection efficiency, which is even faster than 2-gram SVM by combining with GPU computing acceleration. Though 2-gram SVM's training time is shorter than HDN's, this time consumption is not perceived in the real system since the training process is offline. So detection efficiency is actually the key to reflect the efficiency of the methods.

Table 4: The malware family classification results.

Methods	Features	Malware dataset	Malware	Accuracy (%)	Precision	Recall	*F*1-score	TPR (%)	FPR (%)
DroidSIFT [34]	API dependency graph	BD1	1,260	93	-	-	-	-	-
HDN	Opcode patterns	BD1	1,260	*99.04*	0.991	0.989	0.99	97.79	0.5
Drebin [13]	API calls, intents, permissions, cmnds	BD2	5,560	-	-	-	-	93	1
HDN	Opcode patterns	BD2	5,560	98.32	0.986	0.981	0.984	*97.08*	*0.5*

Table 5: The computation efficiency on different methods.

Methods	Training time (h)	Detection time per program (ms)
2-gram SVM	*0.34*	23.62
3-gram SVM	4.18	315.97
Standard LSTM	0.85	*7.74*
HDN (no MBDM)	1.51	14.66
HDN	1.43	14.69

On the other hand, HDN's computing efficiency is relatively lower compared with standard LSTM, since standard LSTM uses truncating and padding strategy, which throws away a large part of sequences, though it increases the computation efficiency, but also leads to a large amount of information loss. By contrast, HDN divides the long sequence into short sequences and processes through the hierarchical structure parallelly, only with a little more time consumption but achieving a much better detection result compared with standard LSTM.

4.3. Visualization Performance for Opcode Embedding. As mentioned in the previous section, HDN can learn opcode semantic information from the dataset through opcode embedding learning, which helps with malware detection task. Here we visualize the trained opcode embedding to explore how the semantic information is saved and represented. We first make dimensionality reduction for the trained opcode embedding by principal component analysis (PCA). It retains a two-dimensional principal component and we use it for visualization in a two-dimensional plane (see in Figure 11).

As we can see, HDN has gained quite a lot of the opcode semantic knowledge through the dataset, and instructions with similar meaning are clustered together. For example, shr-long/2addr&ushr-long/2addr are clustered together at the lower right corner which representing shifting right operation. And const-string&const-class of the upper left corner are clustered together meaning constant values. Besides, long-to-float&float-to-double as a type conversion operation, moniter-enter&moniter-exit which fetches or releases the lock for a specific object, and mul-long&ushr-long as a binary operation performing on two source registers are clustered together. It indicates that HDN learns the semantic relevance of these instructions from data samples. Notably, these instructions are randomly mapped to a vector of length 30 during the initialization training phase, so HDN does not know its text, hence, it cannot capture the similarity through text. Hence, these similarities are mining purely from the opcode sequence context by HDN.

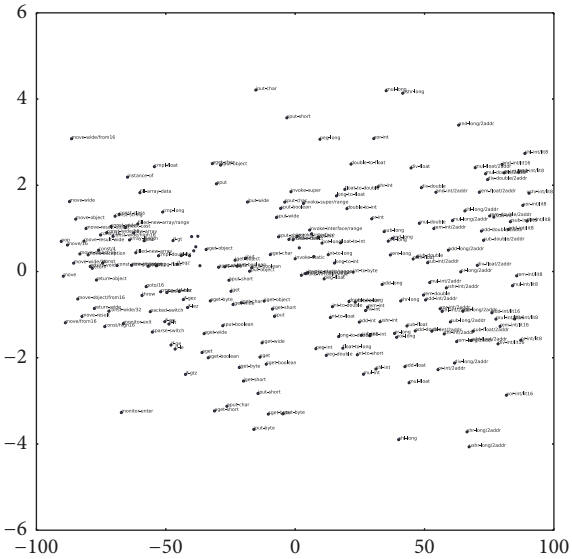

Figure 11: The visual representation of opcode embedding.

5. Conclusion

In this paper, we present a novel malware detection method that uses LSTM-based HDN to learn raw opcode sequence extracted from Android application files which get over the LSTM gradient vanishing problem. For the noisy segments in opcode sequences, HDN uses MBDM for denoising processing. The results show that HDN outperforms other related works on detection accuracy. Also, in comparison with N-gram-based malware detection, HDN can achieve long-range temporal features/patterns mining, greatly reduce

the cost of artificial feature engineering, and still get the better results both on the detection rate and computation efficiency.

However, LSTM-based HDN has some limitation since it does not fully use all related information on Android application, such as permission request and API calls. It is considerable to introduce additional statistical characteristics for HDN to further improve performance. To take advantage of these additional statistical characteristics, one approach is to combine all statistical characteristics into one feature vector as an input to the last logistic regression layer in HDN, along with the features extracted from a hidden layer. Since we mainly focus on designing for HDN methodology rather than implementing an optimal malware detection system, this experiment is not performed.

In the future work, we will mainly carry out the following two aspects of exploration. First, we try to build a flexible and changeable hierarchical network structure, since the current HDN is fixed as a two-level structure which is still difficult to deal with extremely long sequence. So a changeable hierarchical structure will facilitate the processing for different sequence lengths. Second, although LSTM is easy to implement incremental learning, but the reality may encounter a class incremental learning need (such as the appearance of new malware family for malware family detection). It is important to find a way to solve class incremental learning to adapt to the changing malware environment.

Conflicts of Interest

The authors declare that there are no conflicts of interest to this work.

Acknowledgments

This work is partially supported by the National Natural Science Foundation of China under Grants nos. 61672421 and 61402358.

References

[1] W. Enck, P. Gilbert, B. gon Chun et al., "An information-flow tracking system for realtime privacy monitoring on smartphones," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 393–407, 2010.

[2] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang, "Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets," in *Network and Distributed System Security Symposium (NDSS)*, 2012.

[3] L. K. Yan and H. Yin, "Droidscope: seamlessly reconstructing os and dalvik semantic views for dynamic android malware analysis," in *USENIX Security Symposium (USENIX Security)*, 2012.

[4] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *ACM Conference on Computer and Communications Security (CCS)*, pp. 627–638, ACM, Chicago, Ill, USA, October 2011.

[5] D. Barrera, H. G. Kayacik, P. C. Van Oorschot, and A. Somayaji, "A methodology for empirical analysis of permission-based security models and its application to android," in *ACM Conference on Computer and Communications Security (CCS)*, pp. 73–84, USA, October 2010.

[6] H. Peng, C. Gates, B. Sarma et al., "Using probabilistic generative models for ranking risks of android apps," in *ACM Conference on Computer and Communications Security (CCS)*, pp. 241–252, ACM, Raleigh, NC, USA, October 2012.

[7] Q. Jerome, K. Allix, R. State, and T. Engel, "Using opcode-sequences to detect malicious android applications," in *Proceedings of the IEEE International Conference on Communications (ICC '14)*, pp. 914–919, IEEE, Sydney, Australia, June 2014.

[8] K. Chen, P. Wang, Y. Lee et al., "Finding unknown malice in 10 seconds: mass vetting for new threats at the Google-Play scale," in *USENIX Security Symposium (USENIX Security)*, 2015.

[9] Y. Aafer, W. Du, and H. Yin, "DroidAPIMiner: mining API-level features for robust malware detection in android," in *International Conference on Security and Privacy in Communication Networks (SecureComm)*, pp. 86–103, Springer, 2013.

[10] J. Yu, Q. Huang, and C. Yian, "DroidScreening: a practical framework for real-world Android malware analysis," *Security and Communication Networks*, vol. 9, no. 11, pp. 1435–1449, 2016.

[11] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, "RiskRanker: scalable and accurate zero-day android malware detection," in *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services (MobiSys '12)*, pp. 281–294, June 2012.

[12] K. Xu, Y. Li, and R. H. Deng, "ICCDetector: ICC-based malware detection on android," *IEEE Transactions on Information Forensics and Security*, vol. 11, no. 6, pp. 1252–1264, 2016.

[13] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, and K. Rieck, "Drebin: effective and explainable detection of android malware in your pocket," in *Proceedings of the NDSS Symposium 2014*, February 2014.

[14] M. Spreitzenbarth, F. Freiling, F. Echtler, T. Schreck, and J. Hoffmann, "Looking Deeper into Android Applications," in *International Symposium on Applied Computing (SAC)*, pp. 1808–1815, Association for Computing Machinery, March 2013.

[15] L. Cen, C. S. Gates, L. Si, and N. Li, "A probabilistic discriminative model for android malware detection with decompiled source code," *IEEE Transactions on Dependable and Secure Computing*, vol. 12, no. 4, pp. 400–412, 2015.

[16] S. Y. Yerima, S. Sezer, and I. Muttik, "High accuracy android malware detection using ensemble learning," *IET Information Security*, vol. 9, no. 6, pp. 313–320, 2015.

[17] V. Rastogi, Y. Chen, and X. Jiang, "DroidChameleon: Evaluating Android anti-malware against transformation attacks," in *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security, ASIA CCS 2013*, pp. 329–334, China, May 2013.

[18] Y. Du, X. Wang, and J. Wang, "A static android malicious code detection method based on multi-source fusion," *Security and Communication Networks*, vol. 8, no. 17, pp. 3238–3246, 2015.

[19] I. Santos, F. Brezo, J. Nieves et al., "Opcode-sequence-based malware detection," *The Institute of Electrical and Electronics Engineers*, vol. 5965, pp. 35–43, 2010.

[20] I. Santos, F. Brezo, X. Ugarte-Pedrero, and P. G. Bringas, "Opcode sequences as representation of executables for data-mining-based unknown malware detection," *Information Sciences*, vol. 231, pp. 64–82, 2013.

[21] G. Canfora, A. De Lorenzo, E. Medvet et al., "Effectiveness of opcode ngrams for detection of multi family android malware,"

in *International Conference on Availability, Reliability and Security (ARES)*, pp. 333–340, France, August 2015.

[22] R. Pascanu, J. W. Stokes, H. Sanossian et al., "Malware classification with recurrent networks," in *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 1916–1920, Australia, April 2014.

[23] H. Sak, A. Senior, and F. Beaufays, "Long short-term memory recurrent neural network architectures for large scale acoustic modeling," in *Fifteenth Annual Conference of the International Speech Communication Association (ISCA)*, 2014.

[24] R. J. Williams and J. Peng, "An efficient gradient-based algorithm for on-line training of recurrent network trajectories," *Neural Computation*, vol. 2, no. 4, pp. 490–501, 1990.

[25] P. Doetsch, M. Kozielski, and H. Ney, "Fast and robust training of recurrent neural networks for offline handwriting recognition," in *International Conference on Frontiers in Handwriting Recognition (ICFHR)*, pp. 279–284, Greece, September 2014.

[26] K. Chen, Z.-J. Yan, and Q. Huo, "A context-sensitive-chunk BPTT approach to training deep LSTM/BLSTM recurrent neural networks for offline handwriting recognition," in *Proceedings of the 13th International Conference on Document Analysis and Recognition, ICDAR 2015*, pp. 411–415, France, August 2015.

[27] K. Chen and Q. Huo, "Training Deep Bidirectional LSTM acoustic model for LVCSR by a context-sensitive-chunk BPTT approach," *IEEE/ACM Transactions on Audio, Speech and Language Processing*, vol. 24, no. 7, pp. 1185–1193, 2016.

[28] J. Li, M.-T. Luong, and D. Jurafsky, "A hierarchical neural autoencoder for paragraphs and documents," in *The 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (ACL)*, pp. 1106–1115, 2015.

[29] Z. Yang, D. Yang, C. Dyer et al., "Hierarchical attention networks for document classification," in *The 54th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*, pp. 1480–1489, San Diego, Calif, USA, June 2016.

[30] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[31] Y. Zhou and X. Jiang, "Dissecting android malware: characterization and evolution," in *The 33rd IEEE Symposium on Security and Privacy (S & P)*, 2012.

[32] Androzoo., https://androzoo.uni.lu/.

[33] Virustotal, https://www.virustotal.com/.

[34] M. Zhang, Y. Duan, H. Yin et al., "Semantics-aware Android malware classification using weighted contextual API dependency graphs," in *The ACM SIGSAC Conference on Computer and Communications Security (ASIA CCS)*, pp. 1105–1116, ACM, Scottsdale, Ariz, USA, November 2014.

[35] N. McLaughlin, A. Doupé, J. M. Del Rincon, B. J. Kang et al., "Deep android malware detection," in *The Seventh ACM on Conference on Data and Application Security and Privacy (CODASPY)*, pp. 301–308, Scottsdale, Arizona, USA, March 2017.

[36] S. Liang and X. Du, "Permission-combination-based scheme for android mobile malware detection," in *Proceedings of the 2014 1st IEEE International Conference on Communications, ICC 2014*, pp. 2301–2306, Australia, June 2014.

[37] S. Y. Yerima, S. Sezer, G. McWilliams, and I. Muttik, "New android malware detection approach using Bayesian classification," in *Proceedings of the 27th IEEE International Conference on Advanced Information Networking and Applications, AINA 2013*, pp. 121–128, Spain, March 2013.

[38] K. Allix, T. F. Bissyandé, Q. Jérome, J. Klein, R. State, and Y. L. Traon, "Empirical assessment of machine learning-based malware detectors for Android: Measuring the gap between in-the-lab and in-the-wild validation scenarios," *Empirical Software Engineering*, vol. 21, no. 1, pp. 183–211, 2016.

[39] B. Sarma, N. Li, C. Gates, R. Potharaju, C. Nita-Rotaru, and I. Molloy, "Android permissions: a perspective combining risks and benefits," in *Proceedings of the 17th ACM Symposium on Access Control Models and Technologies*, pp. 13–22, ACM, June 2012.

[40] W. Enck, M. Ongtang, and P. McDaniel, "On lightweight mobile phone application certification," in *Proceedings of 16th ACM Conference on Computer and Communications Security*, pp. 235–245, ACM, November 2009.

[41] Libsvm., http://www.csie.ntu.edu.tw/cjlin/libsvm/.