IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received December 4, 2019, accepted January 12, 2020, date of publication January 15, 2020, date of current version February 6, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2966860

A Neural Network-Based Approach for Cryptographic Function Detection in Malware

LI JIA[1], ANMIN ZHOU[1], PENG JIA[1], LUPING LIU[2], YAN WANG[1], AND LIANG LIU[1]
[1]College of Cybersecurity, Sichuan University, Chengdu 610065, China
[2]College of Electronics and Information Engineering, Sichuan University, Chengdu 610065, China

Corresponding author: Anmin Zhou (zhouanmin@scu.edu.cn)

ABSTRACT Cryptographic technology has been commonly used in malware for hiding their static characteristics and malicious behaviors to avoid the detection of anti-virus engines and counter the reverse analysis from security researchers. The detection of cryptographic functions in an effective way in malware has vital significance for malicious code detection and deep analysis. Many efforts have been made to solve this issue, while existing methods suffer from some issues, such as unable to achieve promising results in accuracy, limited by prior knowledge, and have a high overhead. In this paper, we draw on the idea of text classification in the field of natural language processing and propose a novel neural network to detect the type of cryptographic functions. The new network is an end-2-end model which includes two important modules: Instruction-2-vec and K-Max-CNN-Attention. The Instruction-2-vec model extracts the ''words'' of assembly instructions and transfers them into continuous vectors. The K-Max-CNN-Attention is used to encode the instruction vectors and generate the representation of the function. And we designed a softmax classifier to predict the categories of the functions. Extensive experiments were conducted on a collected dataset which contains 15 common types of cryptographic functions extracted from malware, to assess the validity of the proposed approach. The experiment results showed that the proposed approach archives a better performance than the recent embedding network SAFE with the Precision, Recall and F1-score of 0.9349, 0.8933 and 0.9020, respectively. We also compared it with four widely-used tools, the results demonstrated that our approach is much better in accuracy and effectiveness than all of them.

INDEX TERMS Cryptographic function detection, neural network, function embedding, binary analysis.

I. INTRODUCTION

Cryptographic technology plays an important role in protecting data security and integrity of software information security [1]. As a consequence, cryptographic algorithms are also widely used by computer viruses, Trojans and other malware, which leads to growing challenges in information security. They can hide their behaviors and static characteristics such as malicious code and sensitive data, to avoid the detection of anti-virus engines. It also makes the reverse analysis of binary programs from security researchers quite difficult. Therefore, an effective method to detect cryptographic functions in malware and identify their types is necessary. It may be helpful for the researchers to analyze the working principles of malware and extract their program features for deep analysis. Furthermore, it has vital significance for software security analysis and protection of computer system security.

Identifying a certain cryptographic function in a given binary program is a complicated and laborious task [2], especially for malware. Analyzers usually need to use reverse engineering methods, such as disassembly, decompilation, and dynamic debugging, to analyze the assembly code and then try to find out the key information about cryptographic algorithms in programs.

During the past years, plenty of approaches have been proposed to address this issue. These approaches can be mainly divided into two aspects: the static and the dynamic. Static approaches primarily scan against the target binary code or its assembly code to identify and match signatures that may appear in cryptographic algorithms, such as constant

The associate editor coordinating the review of this manuscript and approving it for publication was Amjad Mehmood.

features or instruction features [3]. Dynamic approaches mainly use the various dynamic analysis methods to identify cryptographic functions by dynamically analyzing the running target binary code, according to the instruction operation information during the dynamic execution process. Static approaches usually have high efficiency and are easy to be implemented. Dynamic approaches provide higher accuracy than static approaches, but their efficiency and usability are less effective by the fact that these approaches need to process a large amount of information. Therefore, static approaches are more widely used nowadays. However, traditional static approaches are often based on shallow feature matching, without considering the semantic features of programs; therefore malicious code cannot be recognized if they hide the features of the cryptographic functions deliberately.

With the advance of deep learning, many researchers adopted new methods to detect cryptographic function based on neural networks in recent days. Wright and Manic [4] have used artificial neural network model NNLC to classify functional blocks as being either cryptographic or not, by extracting the frequency of logic instructions (including ''xor'', ''ror'', ''rol'', ''shr'' and ''shl'' instructions) from a disassembled program. The approach FALKE-MC [5] firstly creates classifiers for arbitrary cryptographic algorithms from sample binaries, and then automatically extracts structural features and constants in functions to train a neural network, which can then be applied to locate cryptographic functions in machine code. However, these two methods above can be somewhat divided into traditional static approaches because they still mainly consider extracting features in cryptographic functions.

To address these limitations of existing approaches, in this study, we propose a completely new idea for cryptographic function detection. This new idea comes from the latest studies on the comparison of binary code similarity, where researchers designed deep learning-based binary analysis approaches for this task based on the idea of Natural Language Processing (NLP) [6]–[12]. In these studies, binary programs were expressed in assembly language after being disassembled. Then they treated the instructions as words and designed neural networks to calculate the high-level semantics of the code for similarity comparison. Similarly, in this task of cryptographic function detection, if we can regard an instruction as a word, then it is possible to convert all the instructions into word embeddings. Moreover, it has been experimentally confirmed that NLP technologies are useful for tasks to explore the semantic features of instructions [8], [12].

Inspired by the works mentioned above, in this paper, we proposed a novel solution to detect cryptographic functions based on the instruction semantics of the programs, without any features engineering. In this approach, the ideas and techniques of NLP are borrowed. We argue that different types of cryptographic functions contain their semantic information and can be represented by deep semantic vectors. With the proposed approach, an instruction embedding model named Instruction-2-vec was developed, and an improved neural network K-Max Convolution Neural Network with an Attention mechanism (K-Max-CNN-Attention) was implemented. Firstly, the Instruction-2-vec transfers assembly instructions into 100-dimensional vectors. The K-Max-CNN-Attention is employed to calculate corresponding vector representations for each function, which are fed into the classifier for the detection of cryptographic function and the identification of their type. Besides, a trusted dataset which contains 15 common types of cryptographic functions extracted from malware is generated, and extensive experiments have been conducted to evaluate the proposed approach.

In conclusion, the main contributions of this work are summarized as follows.

- A novel approach for detecting cryptographic functions in malware was proposed. In this process, we regard the program instructions as a semantic text, use the analysis methods in NLP and combine it with a neural network to detect cryptographic functions and identify their types. To the best of our knowledge, this work is the first research to apply the idea of neural network and function embedding to solve the problem of cryptographic function detection in malware.
- K-Max-CNN-Attention is improved on the basis of Dynamic Convolutional Neural Network (DCNN), by employing the parallel convolution approach and adding an attention mechanism. The comparison results with other nine mainstream neural networks indicate its improvement in extracting semantic features.
- The experiment has been conducted to compare the quality of function embeddings produced by our Instruction-2-vec with the newly proposed function embedding architecture SAFE [12]. The experimental results show that our Instruction-2-vec is finer than SAFE and achieves a better performance with the Precision, Recall and F1-score of 0.9349, 0.8933 and 0.9020, respectively.
- The prototype system of the proposed approach has been applied in the real task of cryptographic function detection and classification. Compared to four traditional detection tools, the result with the highest accuracy of 80% proves the effectiveness of the proposed approach.

This paper is organized as follows. Some related works are discussed in Section II. In Section III, we define the problem and give a detail description of the proposed approach. In Section IV, the experiments and results are given. Finally, in Section V we describe the conclusion and future work.

II. RELATED WORK

In this section, prior works are introduced from two perspectives as follows: 1) traditional cryptographic function detection approaches, and 2) deep learning-based binary analysis approaches.

A. TRADITIONAL CRYPTOGRAPHIC FUNCTION DETECTION APPROACHES

Previous work in this area can be divided into two categories: static approaches and dynamic approaches.

1) STATIC APPROACHES

To the best of our knowledge, the problem of cryptographic algorithms identification in binary files was first proposed by Harvey in a research report in 2001 [13]. The report showed that there were different constant characteristics between different types of cryptographic algorithms. For example, most hash functions would use one or several specific constant values, and most block ciphers and serial ciphers would use one or several constant tables. Thus Harvey proposed a simple linear weighted cryptographic algorithm identification method to locate the approximate location of the cryptographic algorithm.

Much of the subsequent work in this area started to build on the field of automated protocol reverse engineering [14]–[20], and mostly adopted feature matching methods. Work in [14], [16]–[18] all took advantage of the fact that bitwise arithmetic instructions accounted for a large proportion of the total instructions in cryptographic functions. Liu *et al.* [21] extracted the characteristic-codes from kinds of cryptographic algorithms and constructed a static characteristic database, and design a scanning tool for the cryptographic algorithms based on the Boyer-Moore matching algorithm. Similarly, Chang *et al.* [22] created a cryptographic algorithm library, including more than 3000 signature characteristics, and proposed a method to recognize cryptographic algorithms based on functions signature recognition.

Most tools commonly used today employ static approaches to determine whether a cryptographic algorithm is presented in a given binary program, such as KANAL plugin [23] for PEiD, Draft Crypto Analyzer (DRACA) [24], Signsrch [25], Crypto Searcher [26], Hash & Crypto Detector (HCD) [27], SnD Crypto Scanner [28], Findcrypt plugin [29] for IDA Pro, and IDAscope [30]. Except that IDAscope is based on features of basic blocks and loop structures, the other tools all take advantage of certain static properties of different algorithms as their signatures.

Static analysis-based detection methods are the most widely used, as a result of the advantages that they are easy to be implemented and usually have high efficiency. However, the limitation of the static approaches is the difficulty of ensuring the soundness and completeness of the pre-constructed cryptographic algorithm features database. Thus they cannot identify cryptographic algorithms that use unknown characteristics or dynamically generate characteristics, which further leads to the problem of low accuracy.

2) DYNAMIC APPROACHES

The dynamic identification of cryptographic function in software is first proposed by Lutz [18]. His approach is based on three features: 1) cryptographic algorithms heavily use integer arithmetic instructions, 2) there are a large number of loops in cryptographic algorithms, and 3) the decryption process decreases information entropy of tainted memory. One of its core methods is to use taint analysis and estimate if a buffer has been decrypted by measuring its entropy, using the dynamic binary instrumentation tool Valgrind on the Linux platform.

Wang *et al.* [19] implemented a dynamic method named ReFormat using data lifetime analysis, including data taint analysis and data flow analysis, to pinpoint the memory locations where messages are decrypted or processed, and then extract decrypted messages from memory. The method is effective for detecting the first cryptographic function in the dynamic instruction execution sequence. Caballero *et al.* [14], [15], [20] refined the methods of Wang *et al.* by identifying several heuristics concerning the number of loop structures and the high percentage of bitwise arithmetic instructions. To enhance detection accuracy, Gröbert *et al.*'s work [2] used PIN tool to dynamically trace the program, and created three heuristics to detect cryptographic basic blocks: chains, mnemonic-const, and verifier based on both generic characteristics of cryptographic implementations and signatures for specific instances of cryptographic algorithms, to enhance the detection accuracy. Zhao *et al.* [1], [31] extended this work using dynamic data pattern analysis, that is, the Input/Output (I/O) correlation of certain ciphers to detect cryptographic data. Besides, the system Aligot proposed by Calvet *et al.* [32] focused on detecting cryptographic algorithms in obfuscated software using loop detection techniques and the I/O relationship comparison.

Dynamic approaches can obtain richer information than static approaches, such as function parameters, API calls, instruction execution traces, instruction encoding codes, and data of memory operations during the dynamic execution [3]. However, the process of dynamic analysis requires a huge consumption, with the recording information may reach to several GB bytes and even much larger [33]. Meanwhile, most dynamic approaches are based on a large amount of prior knowledge for cryptographic algorithms.

B. DEEP LEARNING-BASED BINARY ANALYSIS APPROACHES

In recent years, deep learning has been widely applied to the domain of binary analysis, including function identification and binary code similarity detection.

1) DEEP LEARNING-BASED FUNCTION IDENTIFICATION

Ever since Shin *et al.* [34] firstly applied neural networks to binary analysis in binary code, deep learning has shown stronger results than other traditional approaches. They proposed to use Recurrent Neural Networks (RNN) for the function identification task in binary code, taking bytes of the binary as input and predicting the locations where a function boundary is present. The empirical evaluation results revealed that neural networks could solve the function identification problem more efficiently than the previous methods, which served as an inspiration for further research.

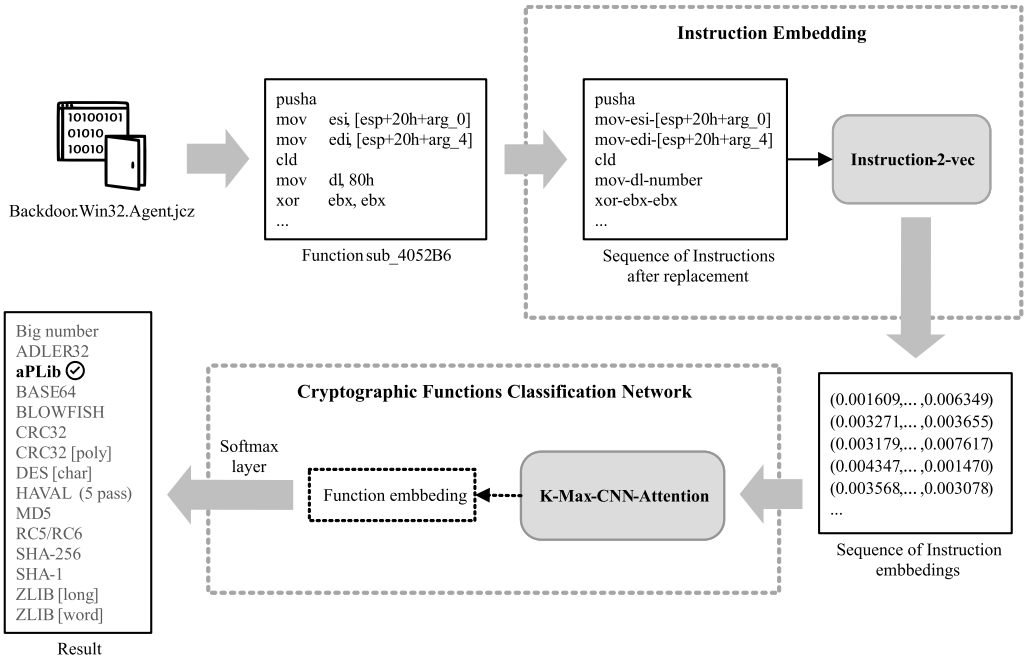

FIGURE 1. The overall architecture of the detection approach.

2) NLP-BASED BINARY CODE SIMILARITY DETECTION

Recently in the field of binary analysis, NLP ideas are widely used to solve the binary code similarity task. The reason is that if we regard instructions as words in NLP problems, then the use of word-embedding of instructions would be particularly useful in binary problems.

The first work that has introduced NLP concepts in the field of binary analysis is EKLAVYA [6], using RNN to learn function signature from disassembled binary codes. Lee *et al.* [7] proposed Instruction2vec for converting assembly instructions to vector representations. Notably, Ding *et al.* [8] proposed a novel approach for assembly clone detection that employed representation learning to construct a feature vector for assembly code. They developed a model called Asm2Vec, which can produce a numeric vector for each function based on embedding the Control Flow Graph (CFG) of functions [35]. Similarly, Xu *et al.* [9] implemented a similarity detection architecture called Gemini, where function embeddings are computed using a graph embedding network Structure2vec [36] based on the CFG of each binary function and the similarity detection can be done by measuring the distance between the embeddings for two functions. Zuo *et al.*'s work [10] solved the task of binary similarity by converting a basic block into an embedding and measuring the distance between two embeddings, and instructions in a basic block are combined through the use of a Long Short-Term Memory (LSTM). Redmond *et al.* [11] proposed a joint learning approach to generating instruction embeddings that capture not only the semantics of instructions within architecture but also their semantic relationships across architectures. SAFE [12], proposed by Massarelli *et al.*, is a general architecture for calculating binary function embeddings starting from disassembled binaries, using a self-attentive recurrent neural network that parses all instructions according to their addresses. The authors also designed another general network architecture for calculating binary function embeddings starting from the corresponding CFG [37].

In summary, with the advantage of neural networks that their parameters can be trained end-to-end and they rely on as little domain knowledge as possible [9], deep learning-based approaches and NLP-based technologies have been proved to be viable to solve problems in binary analysis. In the sight of these studies above, we proposed a novel neural network-based approach for cryptographic function detection in malware.

III. THE PROPOSED APPROACH

A. OVERVIEW

The overall architecture of the detection approach proposed in this paper is shown in Fig. 1.

A given malware is firstly disassembled. For a function inside (e.g. the function "sub_4052B6"), all of its assembly instructions would be extracted and fed to the Instruction Embedding component. These instructions would be converted to a sequence of 100-dimensional embeddings. And then the Cryptographic Functions Classification Network would identify which cryptographic algorithm the function takes.

It is worth mentioning that the proposed approach is verified in a dataset consisting of unpacked and unobfuscated malware samples. For a packed malware, it is a necessary step to try to unpack it automatically or manually before disassembling and extracting the assembly instructions.

There are two primary components:

- **Instruction Embedding.** In this component, the Instruction-2-vec takes a sequence of assembly instructions as input, which have been processed according to some replacement rules. And its output is an embedding matrix containing the vector representation of each instruction in a 100-dimensional space.
- **Cryptographic Functions Classification Network.** Given the instructions represented as vectors, we train a neural network named K-Max-CNN-Attention, feeding it with the sequence of instructions corresponding to several functions to compute the function embeddings. And the Softmax classifier would predict the categories of the functions.

B. INSTRUCTION EMBEDDING

In NLP, one common idea is to map words into vectors that contain numeric values so that the machine can process it. In this paper, we regard the instructions of the program as words, then we employ the Word Embedding technique [38], [39] to convert the instructions into continuous vectors and extract the contextual relationships.

1) INSTRUCTION-2-VEC

Our instruction embedding model is called Instruction-2-vec. The goal of this model is to uncover the semantic information of each instruction from their contextual use in the binary through learning, such as one or group of instruction with similar meaning having similar representation.

Word embedding in our system is implemented by Word2Vec [40] proposed by Mikolov *et al.* [41]. The Continuous Bag-of-Words (CBOW) and the Skip-gram are two representative types of Word2Vec, with relatively low memory use and overall increased efficiency. The CBOW model uses the continuous distributed representation of the context and predicts the current word based on the context, while the Skip-gram model uses each current word as an input and predicts the context within a certain range before and after the current word. During training, a sliding window is applied to a text. Each model starts with a random vector for each instruction and then gets trained when going over each sliding window. After the model is trained, the embeddings of each instruction become meaningful, yielding similar vectors for similar instructions.

The inputs to our Instruction-2-vec are assembly instructions, including both the instruction mnemonic and the operands. An assembly instruction includes one mnemonic (specifying the instruction operation), as well as zero or more operands (specifying registers, memory locations, or literal data) [36]. For example, the mnemonic and operands of the instruction ''mov ebp, esp'' is shown in Fig. 2. Note that the assembly code in this paper adopts the Intel syntax.

However, we do not use the raw assembly instructions directly but laid down a number of rules to replace several operands in instructions. The reason is that some operands which contain values or addresses carry several vital

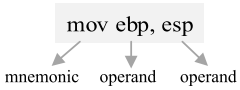

FIGURE 2. The mnemonic and operands of a "mov" instruction.

```
lea   edx, [edx+ecx-28955B88h]        lea   ecx, [eax+ecx-28955B88h]
mov   [ebp+var_10], edx               mov   [ebp+var_4], ecx
mov   eax, [ebp+var_10]               mov   edx, [ebp+var_4]
shl   eax, 7                          shl   edx, 7
mov   ecx, [ebp+var_10]               mov   eax, [ebp+var_4]
shr   ecx, 19h                        shr   eax, 19h
or    eax, ecx                        or    edx, eax
...                                   ...
lea   ecx, [ecx+eax-173848AAh]        lea   edx, [ecx+edx-173848AAh]
mov   [ebp+var_10], ecx               mov   [ebp+var_10], edx
mov   edx, [ebp+var_10]               mov   eax, [ebp+var_10]
shl   edx, 0Ch                        shl   eax, 0Ch
mov   eax, [ebp+var_10]               mov   ecx, [ebp+var_10]
shr   eax, 14h                        shr   ecx, 14h
or    edx, eax                        or    eax, ecx
...                                   ...
lea   eax, [eax+edx+242070DBh]        lea   eax, [edx+eax+242070DBh]
mov   [ebp+var_10], eax               mov   [ebp+var_C], eax
mov   ecx, [ebp+var_10]               mov   ecx, [ebp+var_C]
shl   ecx, 11h                        shl   ecx, 11h
mov   edx, [ebp+var_10]               mov   edx, [ebp+var_C]
shr   edx, 0Fh                        shr   edx, 0Fh
or    ecx, edx                        or    ecx, edx
...                                   ...
lea   edx, [edx+ecx-3E423112h]        lea   ecx, [eax+ecx-3E423112h]
mov   [ebp+var_10], edx               mov   [ebp+var_8], ecx
```

FIGURE 3. Parts of two *MD5* functions.

information to uncover the semantics of instructions but some do not. For example, the *MD5* algorithm is a cryptographic algorithm to calculate a 128-bit hash value. There are parts of two *MD5* functions shown in Fig. 3, and where ''28955B88h'', ''173848AAh'', ''242070DBh'', ''3E423112h'' appeared in ''lea'' instructions in both functions, which may be an important feature to improve the embedding quality. Moreover, we do not keep the address of each instruction as input. Therefore, for the instructions that have useless jump addresses as their operands, no filtering or processing might be counterproductive, probably decreasing the quality of our embeddings.

Our rules for replacement are as follows.

Rule 1: Replace all the immediates with the symbol ''number''. For instance, the instruction ''mov dl, 80h'' becomes ''mov dl, number'', and ''push 0BB8h'' becomes ''push number''.

Rule 2: Replace all the operands which represent addresses with the symbol ''xxx_addr'', including the following three situations:

- ''byte_addr''/''word_addr''/''dword_addr''/''str_addr''/''stru_addr''/''off_addr'': the operand is a memory address of data (e.g. a byte, a string, a struct, and so on). For instance, ''cmp dword_411FDC[eax*4], esi'' becomes ''cmp dword_str[eax*4], esi'', and ''mov edi, offset asc_410B6C'' becomes ''mov edi, offset str_addr''. Note that some instructions whose operands seem not like but actually is an address of data also need to be modified (e.g. ''push offset aTooManyThreads'' is modified to ''push offset str_addr'').

- "sub_addr": the operand is an address of a function, mostly appeared in "call" instructions and sometimes after "offset" in "push" instructions. For instance, "call sub_403BE5" becomes "call sub_addr", "push offset sub_401E30" becomes "push offset sub_addr". Note that there are lots of operands that are directly represented using the function name. We keep the operands named with the standard API unmodified but replace those whose name is long above some threshold or meaningless with the symbol "function" (e.g., "call _memset", "call ds:CreateThread" and "push offset LibFileName" are not modified, while "call ??0CString@@QAE@XZ" is modified to "call function").
- "loc_addr": the operand is an address in the code segment, such as labels of a basic block with prefix "loc_", usually appeared in jump instructions. For instance, "jnz loc_403BDD" becomes "jnz loc_addr".

Note: Besides the above two rules, there is a special rule for numbers that appeared in memory operands. We know that there are two types of memory operands:

- [addr]: instructions with such operands as base memory addresses use the direct addressing mode. We replace this type of operands with the symbol "mem". For example, "mov eax, [434DA5h]" becomes "mov eax, mem". However, this kind of instruction nearly not appeared.
- [reg_base+reg_index*scale+offset]: reg_base is the register saving base memory address, and reg_index is the register saving index values. Among them, reg_base is the necessary one, and the other three are optional. We keep this type of instructions unmodified, such as "mov eax, [esi+34h]", "lea edi, [edi+edx-16493856h]", "mov [esi+ecx*4+10h], eax" and so on.

We train our Instruction-2-vec using the Skip-gram algorithm. The reason is that comparing to another algorithm CBOW, it shows better performance on our dataset and extracts more semantics for each instruction in our experiments (see Section IV-B for details). Finally, the output of our Instruction-2-vec is a map from instructions to a 100-dimensional vector. 100 represents the dimensionality of the instruction vectors, which is set with the reference of the dimension value suggested in [12].

C. CRYPTOGRAPHIC FUNCTIONS CLASSIFICATION NETWORK

Our classification network is implemented using a neural network that we call K-Max-CNN-Attention to carry out the task to classify cryptographic functions. K-Max-CNN-Attention is based on DCNN. The reasons why we considered to adopt K-Max-CNN-Attention are its advantages as follows: 1) its K-Max pooling layer makes it possible to not only extract more than one active feature information in the sentence but preserve the information of their order [42]; and 2) compared to DCNN, it additionally uses the parallel convolution approach and an attention mechanism, making the network more scalable and more efficient to learn richer semantic features.

1) OVERVIEW OF DCNN

Currently, CNN and RNN are two networks that are the most widely used in deep learning. The structure of a CNN model can be normally divided into five layers: input layer, convolutional layer, pooling layer, fully connected layer, and output layer. In briefly, the convolutional layer is an important part of feature extraction for CNN, which learns through convolution kernels and then outputs the feature maps. A convolution layer is followed by a pooling layer, whose role is to count the convolution results, reducing the dimensions. The fully connected layer is used to compress the feature dimensions further to yield the more abstract representation of features. Finally, for classification problems, the output layer would give k outputs if there are k categories.

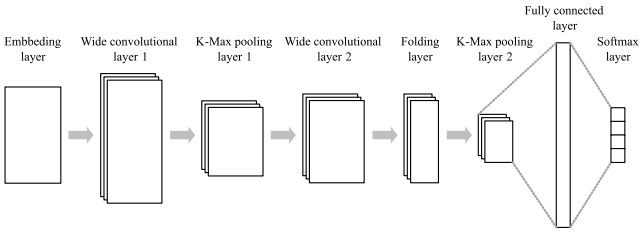

FIGURE 4. The architecture of DCNN.

The pooling layer of CNN models in NLP usually adopt the Max pooling operation, which is to select only one maximum value in each feature map. However, since important feature information may be distributed in many different positions of the sentence, Max pooling has two defects: first, after max pooling, we can obtain the only one most relevant information, and most of the other information would be lost. Second, information on the order of features after Max pooling would be lost completely. Afterward, the method of K-Max pooling was proposed by Kalchbrenner *et al.* [42]. They described a DCNN model, shown in Fig. 4, which has two main innovations: 1) to replace narrow convolution with wide convolution, and 2) to propose the K-Max pooling operation for the first time.

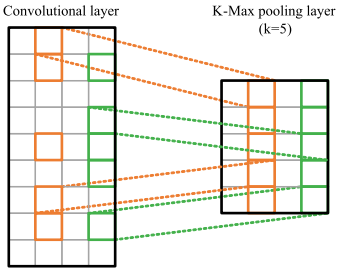

FIGURE 5. A K-Max pooling layer which has values k of 5.

Fig. 5 is a schematic diagram of K-Max pooling, in which each row of the matrix in the wide convolutional layer is

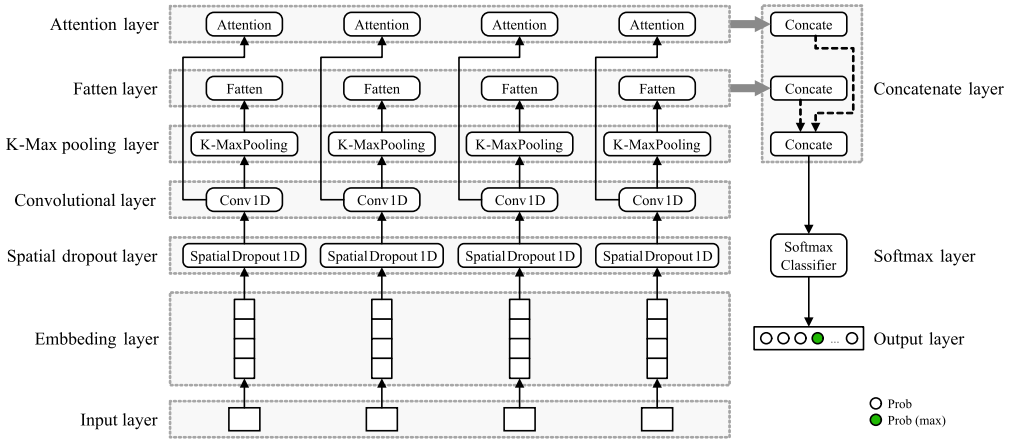

FIGURE 6. The structure of K-Max-CNN-Attention.

a feature map. K-Max pooling selects the largest k values in each row in their original order. Therefore, it becomes possible to not only preserve some position information implicit in the order of features but extract important information of more active features.

2) K-MAX-CNN-ATTENTION

The structure of K-Max-CNN-Attention is shown in Fig. 6.

First, the network is fed with the assembly instruction sequences of functions. They would be transformed into a two-dimensional matrix X consisting of several matrices of instruction vectors using the embedding layer:

$$X = [\vec{x_1}, \vec{x_2}, \cdots, \vec{x_n}] \tag{1}$$

where $\vec{x_i} \in \mathbf{R}^d$ is each instruction vector with d as the instruction embedding dimension, and n is the number of instructions for a function.

The next layer is the spatial dropout layer [43], which is to improve generalization performance by preventing activations from becoming strongly correlated. The layer randomly zeros all pixels of a certain column and then perform a scale transformation on the remaining active pixels.

Then four convolutional layers are used, and each convolutional layer would output a tensor F:

$$F = \text{Conv1D}(X^d) \tag{2}$$

where X^d is the output of spatial dropout layer, $F = [S_1, S_2, \cdots, S_m]$ is the output tensor consisting of m feature maps after a convolution, $S_i \in \mathbf{R}^{n-kernel\_size+1}$ denotes each feature map which can be regarded as a one-dimensional sequence, $kernel\_size$ is the size of convolution kernels, and m is the number of convolution kernels (filter channels). For the four convolutional layers, the values of $kernel\_size$ are set to 5, 6, 7 and 8, respectively, and m are all equal to 128.

Each convolutional layer is followed by a K-Max pooling layer and an attention layer:

- **K-Max pooling layer**. There are m feature maps to be input into this layer. Given a value k and a feature map S of length $p(p \geq k)$, K-Max pooling selects a subsequence of the k highest values of S:

$$\text{KMaxPooling}(S) = \max_k(f_1, f_2, \cdots, f_p) \tag{3}$$

where f_i are features in a feature map S and $p = n - kernel\_size + 1$. The output of K-Max pooling are m subsequences of length k, and the order of the values in each subsequence corresponds to their original order in S. For each K-Max pooling layer, the value of k is set to 3.

- **Attention layer**. The key to the attention mechanism is to learn the weight of a feature distribution, and then use it to fuse the features. We use a simple attention mechanism proposed by [44], computing a weighted average of the different channels across time steps. In a single input channel, the representation vector for the function $\vec{v}$ is found by a weighted summation over all the time steps using the attention importance scores as weights:

$$\vec{v} = \sum_{i=1}^{T} a_i h_i \tag{4}$$

where $h_i(i = 1, 2, \cdots, t)$ are the representation of the instructions and a_i are the attention importance scores for each time step. At time step t_i, the attention importance score a_t is obtained by multiplying the instruction representations with the weight matrix w_a randomly initialized for the attention layer and then normalizing to construct a probability distribution over the words:

$$e_t = h_t w_a \tag{5}$$

$$a_t = \frac{\exp(e_t)}{\sum_{i=1}^{T} \exp(e_i)} \tag{6}$$

Since the parameter m representing the number of convolution kernels as well as channels, the output attention matrix is $W_v \in \mathbf{R}^{m \times 1}$.

The outputs of the K-Max pooling layer would be flattened as a tensor $W_f \in \mathbf{R}^{m\times k}$ using a flatten layer. Then the concatenate layer would complete the integration of the features from different layers by splicing them into one feature tensor M:

$$M = \text{Concate}(\text{Concate}(W_{v_i}), \text{Concate}(W_{f_i})) \tag{7}$$

where $W_{v_i}(i = 1, 2, 3, 4)$ are four attention matrices and W_{f_i} are outputs of four flatten layers.

Finally, the feature tensor M is used as input to the final softmax layer for classification, and there would be several probability values outputted:

$$Prob = \text{Softmax}(M) \tag{8}$$

where each *Prob* indicates the probability that the function adopts a cryptographic algorithm, and the classifier will determine which type of cryptographic function the function belongs to based on the maximum value.

3) THE LOSS FUNCTION

We adopt "Categorical Cross Entropy" (CCE) [45] as the loss function. The definition of the loss function is as follow:

$$Loss = -\sum_{i=1}^{N} y_i \ln \bar{y}_i \tag{9}$$

where y_i denotes the target label for the sample i, $\bar{y}_i$ denotes the model's predicted output for the sample i, and N denotes the number of training samples.

IV. EVALUATION

A. DATASETS AND METRICS

1) DATASETS

Our datasets were generated using malware samples collected from VX Heaven Virus Collection [46]. VX Heaven is a malware collection library website that provides a 47.88GB public dataset with more than 270,000 malware samples, each named after Kaspersky's naming convention. Our processing steps of the datasets are described as follows.

a: SCREENING

We used PEiD to determine if each malicious sample is packed, as well as its compiler information. We subsequently selected 32,067 malicious samples without any software protection technology like packing, encryption or code obfuscation, compiled with the Microsoft Visual C++ compiler, to form our original malicious sample dataset.

b: DISASSEMBLY

We disassembled all 32067 malicious samples in our sample dataset using IDA Pro and extracted their assembly instructions of each sample as a single file.

c: DENOISING

The assembly instruction files extracted from IDA Pro contains the code segment, the data segment, the bss segment, the extern segment, the got/plt segment, and the stack contents of each function. However, only the assembly instructions themselves in the code segment are necessary for us. Contents in other segments, and even the unrelated noise information, including the "," symbols in instructions and a large number of comments, would be removed.

d: LABELING

We selected 7638 malicious samples of the Backdoor type in the dataset and used KANAL plugin for PEiD to identify cryptographic functions in them. After identification, there were 2,959 samples with cryptographic functions covering 90 types of cryptographic algorithms. We filtered out the top 15 types with the largest number of functions, including a total of 2491 cryptographic functions. The statics of the dataset is illustrated in Table 1. Then we successfully extracted the complete assembly instructions of these functions and labeled them with their cryptographic algorithm type.

TABLE 1. The number of functions for 15 types of cryptographic algorithms.

Cryptographic Algorithm Type	Number of Functions
ADLER32	402
aPLib	18
BASE64	224
BLOWFISH	23
CRC32	487
CRC32 [poly]	39
DES [char]	132
HAVAL (5 pass)	145
MD5	450
RC5/RC6	63
SHA-256	26
SHA-1	43
ZLIB [long]	126
ZLIB [word]	151
Big-number	162
Total	**2491**

e: DATASET GENERATION

Our experiments require three sets of datasets for different tasks:

- Dataset 1: to train the Instruction-2-vec model.
- Dataset 2: to train the K-Max-CNN-Attention and evaluate the classification.
- Dataset 3: to test our system and compare the result of detection accuracy with other cryptographic function detection tools.

After completing the "Denoising" step, we obtained several assembly instructions files for 32,067 malicious samples that consisted of the corpus, as well as, Dataset 1. The 2,491 cryptographic functions obtained after the "Labeling" step belong to 15 types. We randomly selected 100 cryptographic functions from a certain range of proportions to make up the Dataset 3, ensuring that there was at least one function

for each type. And the remaining 2391 functions were then composed for Dataset 2. Note that Dataset 2 does not coincide with Dataset 3.

2) METRICS

To evaluate the performance of our model, we used three metrics: Precision, Recall, F1-score. The definitions of them are as follows:

$$Precision = \frac{TP}{TP + FP} \tag{10}$$

$$Recall = \frac{TP}{TP + FN} \tag{11}$$

$$F1_{score} = \frac{2 \times Precision \times Recall}{Precision + Recall} \tag{12}$$

where TP is the number of true-positive predictions, FP is the number of false-positive predictions, and FN is the number of false-negative predictions. And we can conveniently evaluate a model according to the F1-score, the harmonic mean of precision and recall.

B. ENVIRONMENT SETTINGS

Our experiments are conducted on a PC with Intel Xeon E3-1231v3 CPU (8 cores) running at 3.40GHz, 8GB of main memory, 1TB memory, and 256GB SSD, on which Windows 7 Ultimate (64-bit) is installed as the operating system. We implement the prototype system and carry all the experiments out in Python using the Keras library which runs on the Tensorflow backend.

1) TRAINING INSTRUCTION-2-VEC

We train the Instruction-2-vec using the Skip-gram algorithm. As was introduced in Section III-B, the Skip-gram model uses the current instruction to predict the instructions around it while CBOW predicts the current instruction using the instructions around it. The experimental result of Mikolov *et al.* [41] shows that if the trained dataset is large enough, the effect of the Skip-gram model is better than that of the CBOW model.

To compare the quality of the two different versions of instruction vectors, we trained the Instruction-2-vec using both CBOW and Skip-gram algorithms, respectively, with the same training corpus Dataset 1 and the same dimensionality of the instruction vectors. We set the size of the sliding window 5, which indicates the maximum distance between the current word and the predicted word in a sentence while training. Moreover, the dictionary is allowed to be truncated; that is, words with word frequency less than five will be discarded.

Previous papers in NLP typically use a table showing example words and their most similar words, and understand them intuitively. However, it is not feasible for us because we cannot judge which type of instruction vectors get better semantics for our task by simply dividing one instruction similar to another instruction. The two instruction embedding models were connected to the same simple CNN classification network to measure the quality of their instruction vectors.

The comparison of two instruction embedding models from CBOW and Skip-gram algorithms are shown in Table 2. Compared to CBOW algorithms, Skip-gram still has a higher classification accuracy on our corpus dataset, extracting more semantic information suitable for our tasks and showing better performance, besides, with the training time a little bit shorter.

TABLE 2. Comparison between CBOW and Skip-gram algorithms.

Items	CBOW	Skip-gram
Length	178217	178217
Training time	4h13m5s	4h12m50s
Accuracy	0.9333	0.9417

2) TRAINING CLASSIFICATION NETWORK

We split Dataset 2 into three disjoint subsets of functions for training, validation, and testing respectively in the proportion of 70%-15%-15%. Besides, it is guaranteed that no one function is separated into two different subsets. In doing so, it can be examined whether the pre-trained model can generalize to unknown functions.

For training, there are many hyper-parameters of the model that need to be set, which are tuned on the validation subset. As was introduced in Section III-C, the embedding dimension is 100. The number of convolution kernels (filters) of 4 convolutional layers is 128, and the kernel size is 5, 6, 7, and 8, respectively. The value of k for each K-Max pooling layer is 3. The optimal values of all the hyper-parameters are given in Table 3.

TABLE 3. Hyper-parameters.

Items	Value
Embedding dimension	100
Spatial dropout rate	0.2
Filters (Convolution kernels)	128
Kernel size of 4 convolutional layers	5, 6, 7, 8
Value of k for K-Max pooling layers	3
Dropout rate	0.6
Epochs	100
Mini-batch size	32
Learning rate	0.001
Sentence length	1100
Optimizer	Adam

We trained the model for 100 epochs (an epoch represents a complete pass over the whole training set). In order to get a better generalization performance, an early stopping method was adopted to prevent overfitting. The performance of the model was calculated on the validation subset in each epoch while fitting the model. If the loss of the current epoch on the verification subset is lower than that of the previous epoch, then the current model weight is saved; otherwise, after 5 more epochs, if the performance is still not better,

TABLE 4. Comparison of training time per epoch and evaluation results among ten networks. The 2nd column *Epochs* is the number of epochs that each network has actually trained when the early stopping method terminated the training process.

Networks	Epochs	Trainning Time	Time/Epoch	Precision	Recall	F1-Score
BiLSTM	39	34m58s	53.79s	0.5265	0.5655	0.5236
CNN_LSTM	39	23m27s	36.08s	0.5382	0.5710	0.5247
RCNN	58	55m19s	57.22s	0.9383	0.9136	0.9135
AvRNN	56	1h46m8s	113.71s	0.9368	0.9304	0.9294
DropoutAvRNN	66	2h7m20s	115.76s	0.9162	0.8997	0.8950
DropoutBiGRU	47	1h40m11s	127.89s	0.9011	0.8969	0.8879
textCNN	30	27m16s	54.53s	0.9272	0.9248	0.9216
AvCNN	66	1h30m34s	82.33s	0.9471	0.9276	0.9270
K-Max-CNN	62	1h18m18s	75.77s	0.9586	0.9499	0.9499
K-Max-CNN-Attention	**82**	**1h12m23s**	**52.96s**	**0.9626**	**0.9582**	**0.9569**

training would be stopped and the last saved weight would be used as the weight of the final model. Besides, we used a mini-batch size of 32, the learning rate of 0.001, and the Adam optimizer. And the sentence length threshold was set to 1100, that is to say, if the number of instructions inside each function is greater than 1100, the extra part will be cut off. There are two reasons: 1) we believe that length-truncation can guarantee a better balance between training time and accuracy, and 2) the length of the great majority of functions in our datasets is below this threshold. The statistics are presented in Fig. 7, showing that there are 90% of the 2491 functions having less than 1093 instructions.

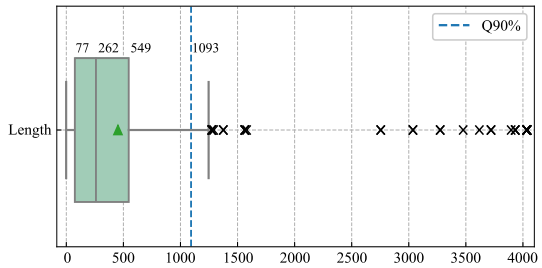

FIGURE 7. Boxplot for length (the number of instructions) of 2491 functions.

C. EXPERIMENT 1 - COMPARISON OF NETWORKS

In our case, we compared the K-Max-CNN-Attention with several baseline CNN structures like textCNN [47], AvCNN, K-Max-CNN, and tried some other neural network structures, such as BiLSTM, CNN_LSTM, RCNN, AvRNN, Dropout-BiGRU, and DropoutAvRNN.

1) METHODOLOGY

We kept these models having the same input layer, embedding layer, and softmax layer and fine-tune the other layers in the network to the target task until convergence on our validation subset of Dataset 2. To examine how the accuracy of the model fluctuated during training, we set the maximum training epoch for each model to 100 and validated it every epoch for the accuracy and loss. Note that the early stopping method is still adopted. After fitting the model, we evaluated it with the testing subset, and three metrics would be calculated: Precision, Recall, F1-score.

2) EXPERIMENTAL RESULTS

The experimental results on all models are displayed in Table 4 and Fig. 8.

From Table 4, it can be seen that all the networks stopped early as a result of the early stopping method. Most of them perform very well except BiLSTM and CNN_LSTM. However, it is obvious that K-Max-CNN-Attention shows the best performance on the evaluation of all these ten models.

From Fig. 8(a) and Fig. 8(b), it can be observed that during the first few epochs, the loss value decreased rapidly and the accuracy value increased greatly. After about 20 epochs, the loss value kept a small decrease and gradually approached 0, and it finally stayed at 0.0722 when the training was over. At the same time, the accuracy value has been continuously increasing slightly and gradually remaining stable.

From Fig. 8(c) and Fig. 8(d), the trend of the accuracy values and the loss values on the validation subset after each epoch is very similar to the training process, with some small fluctuations. The highest validation accuracy was 0.9582, which occurred in the 76th epoch, and the subsequent 6 epochs had a small drop, but the best model weight has been saved because of the early stopping method.

3) TIME CONSUMPTION CONSIDERATIONS

We counted the training time and numbers of epochs in the entire training process for each network, see in Table 4. The result showed that among the remaining nine networks except CNN_LSTM, K-Max-CNN-Attention spent the shortest training time in each epoch with the highest F1-scores 0.9569. Furthermore, although CNN_LSTM took the shortest time, it only achieved F1-scores at 0.5247.

To sum up, it is believed that our K-Max-CNN-Attention model can be quickly trained to achieve good performance.

D. EXPERIMENT 2 - COMPARISON OF EMBEDDINGS

In this experiment, we compared the system with the recent SAFE, similar to our work, which is an architecture for the cross-platform embedding of functions based on a self-attentive neural network. The reasons why we choose to make a comparative study with SAFE are as follows.

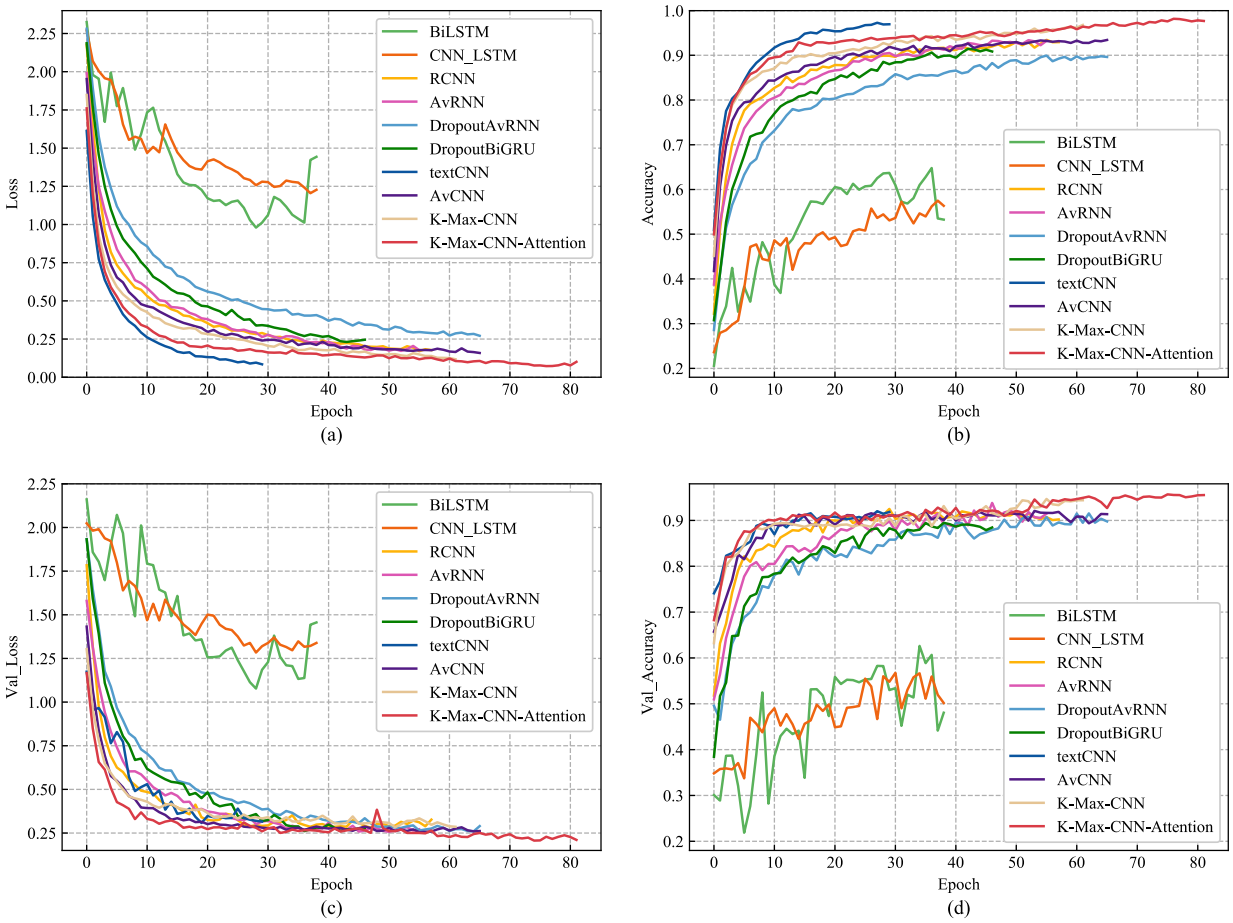

FIGURE 8. Results of loss and accuracy on each epoch for 10 networks. (a) Training loss, (b) training accuracy, (c) validation loss, and (d) validation accuracy.

- SAFE is an end-to-end system from binary file to function embedding, which is one of the most recent studies focusing on the relationship between embedding vectors and the semantics in the domain of binary analysis. SAFE receives binaries and disassembles them using radare2, then computes and finally outputs the 100-dimensional embedding of the function whose starting address need to be provided in advance.
- Although SAFE is designed mainly to solve the binary similarity problem, it is computationally more efficient, as well as has a greater speed advantage, than existing neural network-based solutions for computing function embeddings, such as Gemini. SAFE provided strong confirmation of the crucial assumption that our approach adopts; that is, semantic information in assembly instructions can be represented by deep semantic vectors.
- In its evaluation task of semantic classification (Task 4), SAFE acted very well to identify encryption functions from known malware: among ten functions flagged by SAFE as Encryption, only one was a false positive [12]. This is similar to our task. Moreover, SAFE has been distributed publicly [48].

As a result, it is feasible and credible that the embeddings calculated by our Instruction-2-vec are compared with the ones by SAFE. What is more, the critical difference is that SAFE did not give a further classification on what type of cryptographic algorithms a function took.

1) METHODOLOGY

It is worth noting that the input and output of our system and SAFE are both different. We receive all the assembly instructions of functions that have been pre-extracted. After calculating the embeddings, the result of the classification would be directly output using the softmax layer in K-Max-CNN-Attention. However, the input of SAFE is a binary file and the starting address of a specified function, and the output is the function embedding which integrated the assembly process and computational process together. For this reason, we are unable to take the same testing methodology.

We used Dataset 2 to accomplish this task. However, it was found that there are 182 functions among Dataset 2 that could not be disassembled and extracted by SAFE. Consequently, to guarantee the consistency of the dataset, a new dataset consisting of the remaining 2209 functions was generated.

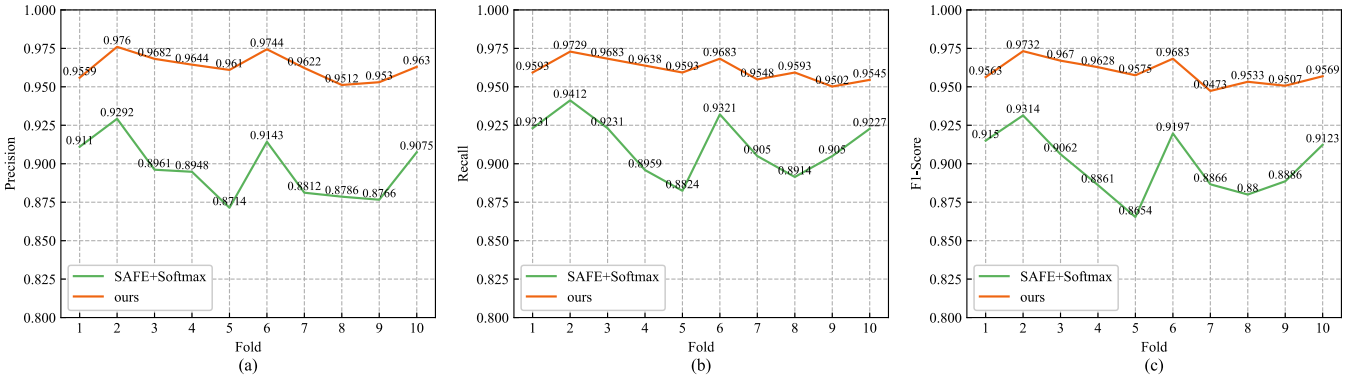

FIGURE 9. Comparison results of weighted metrics on ten folds between safe and our model. (a) Precision, (b) Recall, and (c) F1-score.

Firstly, a softmax classifier was created for SAFE, which was the same as the last softmax layer in K-Max-CNN-Attention, with the same parameters. Given the embeddings of all functions in the new dataset pre-computed using SAFE, we used this softmax classifier for classification. As for ours, we ran K-Max-CNN-Attention to train a classification model on the new dataset. We used the 10-fold cross-validation approach to measure these two models, firstly sorting the new dataset with 2209 functions in advance to ensure that we can test on the same data in two different ways. In each fold, the dataset would be divided into ten parts, one was retained as the testing subset, and the other nine parts were used for the training subset. The number of testing subsets was 221 for the first nine folds and 220 for the tenth, respectively. After executing ten rounds in this way, we can get ten models. Finally, we averaged the performance metrics of the ten models as the overall performance metrics, thereby the quality of the embedding generated by these two methods can be judged. In addition, since there was no validation subset in the cross-validation, we adjusted the quantity to be monitored from the validation loss to the accuracy for the early stopping method we adopted in the training process.

2) EXPERIMENTAL RESULTS

Both SAFE and ours can show a good performance in the task to classify cryptographic functions using function embeddings. Obviously, Fig. 9 shows that our embeddings perform better. Our model consistently obtains F1-scores in the range of 94-98%, while SAFE obtains about 86-94%.

The results of classification performances on all the 15 types of cryptographic algorithms are reported in Table 5. There are 11 types getting an F1-score above 94%, and most are higher than SAFE, except on *RC5/RC6* and *ZLIB [long]*, where SAFE has higher recalls than ours, which indicates that SAFE does better in returning most of the relevant results. Besides, the obtained F1-scores on *BASE64* and *Big-number* are higher than SAFE but less than 90%, though ours obtains much higher precisions meaning that more relevant results are returned than irrelevant. Besides, *SHA-1* is a type that our embeddings perform well while SAFE not. Note that whether it is ours or SAFE, of which the performances on *BLOWFISH* and *CRC32 [poly]* are both not good enough, we speculate that it is probably due that the number of functions belonging to these two types are both less than 20, which makes it impossible for no matter SAFE or ours to extract enough instructions semantic information from such a limited number of functions, which in turn leads to the inability to obtain sufficiently effective features to be used for identifying these two type of cryptographic functions. In summary, most of the

TABLE 5. Results of classification performances on each type of cryptographic algorithm. The 1st column *Type* is 15 types of cryptographic algorithms, and the 2nd column *Support* is the number of functions corresponding to each cryptographic algorithm in the dataset with 2209 functions.

Type	Support	Model	Precision	Recall	F1-Score
ADLER32	372	SAFE+Softmax	0.9676	1.0000	0.9832
		Ours	0.9899	1.0000	0.9948
aPLib	16	SAFE+Softmax	1.0000	1.0000	1.0000
		Ours	1.0000	1.0000	1.0000
BASE64	203	SAFE+Softmax	0.8043	0.6749	0.7295
		Ours	0.8462	0.9064	0.8702
BLOWFISH	17	SAFE+Softmax	0.0588	0.0588	0.0588
		Ours	0.8235	0.6470	0.7059
CRC32	454	SAFE+Softmax	0.9076	0.9515	0.9284
		Ours	0.9803	0.9736	0.9768
CRC32 [poly]	19	SAFE+Softmax	0.0000	0.0000	0.0000
		Ours	0.6842	0.2632	0.3534
DES [char]	117	SAFE+Softmax	0.9083	0.9316	0.9178
		Ours	0.9419	0.9658	0.9507
HAVAL (5 pass)	135	SAFE+Softmax	0.9926	0.9926	0.9926
		Ours	1.0000	0.9926	0.9961
MD5	409	SAFE+Softmax	0.8755	0.9976	0.9321
		Ours	0.9856	0.9902	0.9878
RC5/RC6	60	SAFE+Softmax	0.9762	0.9500	0.9583
		Ours	0.9857	0.9333	0.9558
SHA-256	24	SAFE+Softmax	0.9583	0.8333	0.8708
		Ours	1.0000	1.0000	1.0000
SHA-1	40	SAFE+Softmax	0.7000	0.2750	0.3826
		Ours	0.9315	0.9750	0.9428
ZLIB [long]	118	SAFE+Softmax	0.9610	0.9915	0.9755
		Ours	0.9757	0.9746	0.9744
ZLIB [word]	144	SAFE+Softmax	0.9656	0.9514	0.9575
		Ours	1.0000	1.0000	1.0000
Big-number	81	SAFE+Softmax	0.8244	0.7901	0.7971
		Ours	0.8792	0.7778	0.8218
Average	2209	SAFE+Softmax	0.7934	0.7599	0.7656
		Ours	**0.9349**	**0.8933**	**0.9020**

time, ours performs better than SAFE with F1-scores much higher on over 13 types and a bit lower but very close on the remaining 2 types.

3) SPEED CONSIDERATIONS

We also considered comparing the speed between ours and SAFE. As we did in Experiment 1, the dataset with 2209 samples was split into three subsets for training, validation, and testing. After training for 100 epochs using SAFE added a softmax classifier and K-Max-CNN-Attention, we obtained two models to classify all the functions based on their cryptographic algorithm. The total time spent by SAFE and ours are 4h20m16s and 53m8s, respectively. Unfortunately, it is unreasonable to perform a direct comparison on the speed of the two directly through their total time consumption, since SAFE needs to not only compute the embedding of functions and classify them but also extra disassembly binaries and extract functions.

However, what we additionally found was that compared to SAFE, our model is able to improve by about 6% in F1-score on the dataset containing 2209 samples. The classification results are shown in Table 6.

TABLE 6. Comparison of the classification results on the dataset containing 2209 samples.

Networks	Precision	Recall	F1-Score
SAFE+Softmax	0.9017	0.9066	0.8988
Ours	**0.9607**	**0.9578**	**0.9586**

E. EXPERIMENT 3 - COMPARISON WITH PREVALENT TOOLS

To measure the effectiveness of the proposed approach, we designed a comparison experiment on detection accuracy among ours and several other cryptographic function detection tools, including Findcrypt, IDAscope, HCD, Crypto Searcher and DRACA, and the recognition results of PEiD KANAL are taken as a reference.

1) METHODOLOGY

The evaluation is conducted on Dataset 3, which contains 100 cryptographic functions belonging to 15 types of cryptographic algorithms. Firstly we tested these above tools respectively and calculated which types of cryptographic functions/algorithms each tool supports. The results are shown in Table 7.

For some of the cryptographic algorithms, such as *BASE64*, *BLOWFISH*, *CRC32*, *DES [char]*, *SHA-256*, *ZLIB [long]*, *ZLIB [word]* and *Big-number*, these tools cannot identify the cryptographic functions directly. For example, the *BLOWFISH* algorithm is a 64-bit block cipher algorithm that uses two fixed source key boxes, one is a P-Box consisting of 18 elements, and the other one is an S-Box consisting of 1024 elements (e.g. 0D1310BA6h, 98DFB5ACh, 2FFD72DBh, 0D01ADFB7h and so on). These elements

TABLE 7. Cryptographic algorithm types each tool supports.

Type	Ours	Findcrypt	IDAscope	HCD	Crypto Searcher	DRACA
ADLER32	○	×	○	○	×	×
aPLib	○	×	×	×	×	×
BASE64	○	○	×	○	○	×
BLOWFISH	○	○	○	○	○	○
CRC32	○	○	○	○	○	○
CRC32 [poly]	○	○	○	○	×	×
DES [char]	○	×	○	○	○	○
HAVAL (5 pass)	○	×	○	○	×	×
MD5	○	○	○	○	○	○
RC5/RC6	○	×	○	○	○	○
SHA-256	○	×	×	○	○	×
SHA-1	○	○	×	○	×	○
ZLIB [long]	○	×	○	○	×	×
ZLIB [word]	○	×	○	○	×	×
Big-number	○	○	×	×	×	×

are sequentially stored in memory; thus these static tools generally detect the *BLOWFISH* algorithm by locating the memory that stores the S-Box and P-Box. For these types of cryptographic algorithms above, we believe that as long as the tool can locate the constant values, or constant tables, or other static features that the functions use, the recognition is correct.

For the other cryptographic algorithms, such as *aPLib*, *ADLER32*, *CRC32 [poly]*, *HAVAL (5 pass)*, *MD5*, *RC5/RC6*, and *SHA-1*, only the tool accurately locates an instruction inside the cryptographic function, can we consider that the recognition of the cryptographic function is successful. For example, the *SHA-1* algorithm is also a mainstream hash algorithm with the same message grouping and padding methods as *MD5*, and it uses a series of constants, expressed in hexadecimal form as 5A827999h, 6ED9EBA1h, 8F1BBCDCh, and 0CA62C1D6h. However, they are not stored in memory as *MD5* does but directly appeared in the operand of an assembly instruction inside the cryptographic function, such as "lea edi, [edi+edx+5A827999h]" and "lea ebx, [ebx+ecx+6ED9EBA1h]". Besides, these tools usually detect the *aPLib* functions by identifying an instruction "cmp eax, 7D00h".

According to Table 7, all these six tools including ours support the effective detection of *BLOWFISH*, *CRC32*, and *MD5* functions. Note that for the type of encryption function that supports detection, Findcrypt, IDAscope, and HCD can give the VA (=ImageBase+Offset+VRk) of the recognized instruction or memory, and Crypto Searcher shows ImageBase and Offset respectively. As for DRACA, only the possibility of the cryptographic algorithm exists in a binary can be given. Therefore, we can only compare the detection accuracy of ours with Findcrypt, IDAscope, HCD, and Crypto Searcher.

2) EXPERIMENTAL RESULTS

Table 8 presents the detection accuracy results of each tool. The result shows that the detection accuracy of our model is 80%, which is the highest of all methods. Findcrypt achieves

TABLE 8. Detection accuracy results of five different cryptographic function detection tools. The 2nd column *Support* is the number of functions corresponding to each cryptographic algorithm in Dataset 3.

Type	Support	Ours	Findcrypt	IDAscope	HCD	Crypto Searcher
ADLER32	15	9	0	14	13	0
aPLib	1	1	0	0	0	0
BASE64	9	8	9	0	8	8
BLOWFISH	1	1	1	1	0	1
CRC32	20	16	20	20	17	0
CRC32 [poly]	2	2	2	2	1	0
DES [char]	5	3	0	3	1	1
HAVAL (5 pass)	8	8	0	1	8	0
MD5	20	20	20	1	19	19
RC5/RC6	2	1	0	2	2	2
SHA-256	1	0	0	0	0	0
SHA-1	1	1	1	0	0	0
ZLIB [long]	4	2	0	3	0	0
ZLIB [word]	4	4	0	4	0	0
Big-number	7	4	5	0	0	0
Accuracy	-	**80%**	59%	51%	69%	31%

a good detection accuracy on among those cryptographic function types that it supports to detect, for instance, it reaches a 100% accuracy for detecting *BASE64*, *CRC32* and *MD5* functions. However, among eight *HAVAL (5 pass)* hash functions, Findcrypt incorrectly recognizes seven as *BLOWFISH* functions. IDAscope does well in detecting *ADLER32* and *CRC32*, getting a higher accuracy than ours, but the detection accuracy in *HAVAL (5 pass)* and *MD5* are relatively low. Crypto Searcher performs well in *BASE64* and *MD5* functions detection, however, with low overall accuracy, and it only supports identifying a few types. One thing worth mentioning is that all the Offset given in the results of detecting *CRC32* are wrong values. And HCD is the second most accurate tool besides ours, having an accuracy of 69%, but we noticed that it explicitly supports the detection of *BLOWFISH*, *SHA-256*, *SHA-1*, *ZLIB [long]* and *ZLIB [word]* functions, but actually none of them are successfully identified. We manually analyzed some examples of failure detection, and found that there are five functions, including two *BASE64* functions, one *BLOWFISH* function, one *CRC32* function, and one DES function, the reason why HCD failed to detect them is that although the Offset of the instruction or constant table in the malicious binaries are correctly located, it finally miscalculates their VA. What is more, it is found in this experiment that HCD performed unstably during detection.

Ours performs reliably in detecting most cryptographic function types. Note that there are several cryptographic algorithm types that contain only one or two few support functions, such as *aPLib*, *BLOWFISH*, *CRC32 [poly]*, *RC5/RC6*, *SHA-256*, and *SHA-1*. We manually analyzed these cryptographic functions, especially the *aPLib* and the *SHA-256*, because just ours detected the *aPLib* function, and no tool detected the *SHA-256* function. Moreover, we confirmed that ours only makes an error while detecting the *SHA-256* function. The *SHA-256* function is at address 0x4104F0h

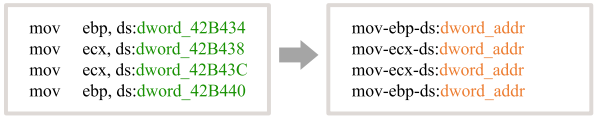

FIGURE 10. Instructions of an *SHA-256* function after the replacement of the operands.

in the malware Backdoor.Win32.Agent.ixi, there are several "mov" instructions calling 64 constants (e.g. 428A2F98h, 71374491h, 0B5C0FBCFh, and so on) which are stored at addresses started on 0x42B434h. However, we guess that because our Instruction-2-vec has replaced these operands in "mov" instructions all as "dword_addr", several special semantic information which may be very useful has disappeared, seen in Fig. 10.

V. CONCLUSION AND FUTURE WORKS

In this paper, we proposed a novel approach based on neural networks for detecting cryptographic functions in malware and have implemented a prototype system. Our architecture consists of two main components: one is Instruction-2-vec that extracts the semantic information of assembly instructions and transfers them into 100-dimensional vectors, and the other one is an improved neural network K-Max-CNN-Attention to calculate function embeddings and carry out the task of classifying the cryptographic functions. The approach was evaluated to be effective from three aspects, such as the performance of K-Max-CNN-Attention, the quality of function embeddings computed by Instruction-2-vec, and the detection accuracy of our prototype in the real use case for cryptographic function detection. The proposed approach indicates promising results that it not only provides an improvement in performance but also achieves high detection accuracy, which is superior to four widely-used traditional cryptographic function detection tools.

Although our approach has shown excellent performance compared to the traditional cryptographic function detection approaches, the detection accuracy remains to be improved. What we plan to do in future work is, firstly, to improve our instruction embedding model. Instructions in a function are not always executed sequentially, so treating a function as a straight-line instruction sequence is still inadequate. We believe that adding the address of the assembly instructions themselves while building the corpus, instead of just retaining the mnemonics and operands as what we do now, may bring about an improvement in the quality of the instruction embeddings. Furthermore, we would like to expand the size of the datasets and use Graph Convolutional Neural Network (GCN) to retrain our classification model so that it is possible to obtain more features while computing function embeddings from information that may implicitly exist in the jump relationships of the basic blocks of functions.

REFERENCES

[1] R. Zhao, D. Gu, J. Li, and R. Yu, "Detection and analysis of cryptographic data inside software," in *Proc. 14th Int. Conf. Inf. Secur. (ICS)*. Berlin, Germany: Springer, 2011, pp. 182–196.

[2] F. Gröbert, C. Willems, and T. Holz, "Automated identification of cryptographic primitives in binary programs," in *Proc. 14th Int. Conf. Recent Adv. Intrusion Detection (RAID)*. Berlin, Germany: Springer, 2011, pp. 41–60.

[3] J. Li, "Research on key technology of cryptography algorithm recognition and analysis," Ph.D. dissertation, Inf. Eng. Univ., Zhengzhou, China, 2014.

[4] J. L. Wright and M. Manic, "Neural network approach to locating cryptography in object code," in *Proc. 14th IEEE Int. Conf. Emerg. Technol. Factory Autom. (ETFA)*, Sep. 2009, pp. 1–4.

[5] A. Aigner, "FALKE-MC: A neural network based approach to locate cryptographic functions in machine code," in *Proc. ACM 13th Int. Conf. Availability, Rel. Secur. (ARES)*, New York, NY, USA, 2018, pp. 2-1–2-8.

[6] Z. L. Chua, S. Shen, P. Saxena, and Z. Liang, "Neural nets can learn function type signatures from binaries," in *Proc. 26th USENIX Secur. Symp. (USENIX Secur.)*, 2017, pp. 99–116.

[7] Y. J. Lee, S.-H. Choi, C. Kim, S.-H. Lim, and K.-W. Park, "Learning binary code with deep learning to detect software weakness," in *Proc. 9th Int. Conf. Internet (ICONI)*, 2017.

[8] S. H. Ding, B. C. Fung, and P. Charland, "Asm2Vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization," in *Proc. 40th IEEE Symp. Secur. Privacy (S P)*, May 2019.

[9] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *Proc. 24th ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*, New York, NY, USA, 2017, pp. 363–376.

[10] F. Zuo, X. Li, Z. Zhang, P. Young, L. Luo, and Q. Zeng, "Neural machine translation inspired binary code similarity comparison beyond function pairs," 2018, *arXiv:1808.04706*. [Online]. Available: https://arxiv.org/abs/1808.04706

[11] K. Redmond, L. Luo, and Q. Zeng, "A cross-architecture instruction embedding model for natural language processing-inspired binary code analysis," 2018, *arXiv:1812.09652*. [Online]. Available: https://arxiv.org/abs/1812.09652

[12] L. Massarelli, G. A. Di Luna, F. Petroni, R. Baldoni, and L. Querzoni, "SAFE: Self-attentive function embeddings for binary similarity," in *Proc. 16th Int. Conf. Detection Intrusions Malware, Vulnerability Assessment (DIMVA)*. Cham, Switzerland: Springer, 2019, pp. 309–329.

[13] I. Harvey, "Cipher hunting: How to find cryptographic algorithms in large binaries," NCipher Corp. Ltd., Cambridge, U.K., Tech. Rep., 2001, pp. 46–51.

[14] J. Caballero, H. Yin, Z. Liang, and D. Song, "Polyglot: Automatic extraction of protocol message format using dynamic binary analysis," in *Proc. 14th ACM Conf. Comput. Commun. Secur. (CCS)*, New York, NY, USA, 2007, pp. 317–329.

[15] J. Caballero, N. M. Johnson, S. McCamant, and D. Song, "Binary code extraction and interface identification for security applications," in *Proc. 17th Netw. Distrib. Syst. Secur. Symp. (NDSS)*, San Diego, CA, USA, Feb. 2010.

[16] Z. Lin, X. Jiang, D. Xu, and X. Zhang, "Automatic protocol format reverse engineering through context-aware monitored execution," in *Proc. 15th Netw. Distrib. Syst. Secur. Symp. (NDSS)*, San Diego, CA, USA, vol. 8, Feb. 2008, pp. 1–15.

[17] G. Wondracek, P. M. Comparetti, C. Kruegel, E. Kirda, and S. S. S. Anna, "Automatic network protocol analysis," in *Proc. 15th Netw. Distrib. Syst. Secur. Symp. (NDSS)*, San Diego, CA, USA, vol. 8, Feb. 2008, pp. 1–14.

[18] N. Lutz, "Towards revealing attacker's intent by automatically decrypting network traffic," M.S. thesis, ETH Zürich, Zürich, Switzerland, 2008.

[19] Z. Wang, X. Jiang, W. Cui, X. Wang, and M. Grace, "Reformat: Automatic reverse engineering of encrypted messages," in *Proc. 14th Eur. Conf. Res. Comput. Secur. (ESORICS)*. Berlin, Germany: Springer, 2009, pp. 200–215.

[20] J. Caballero, P. Poosankam, C. Kreibich, and D. Song, "Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering," in *Proc. 16th ACM Conf. Comput. Commun. Secur. (CCS)*, New York, NY, USA, 2009, pp. 621–634.

[21] T.-M. Liu, L.-H. Jiang, H.-Q. He, J.-Z. Li, and X. Yu, "Researching on cryptographic algorithm recognition based on static characteristic-code," in *Proc. Int. Conf. Secur. Technol. (SecTech)*. Berlin, Germany: Springer, 2009, pp. 140–147.

[22] R. Chang, L. Jiang, H. Shu, and H. He, "Cryptographic algorithms analysis technology research based on functions signature recognition," in *Proc. 10th IEEE Int. Conf. Comput. Intell. Secur. (CIS)*, 2014, pp. 499–504.

[23] Snaker. *KANAL—Krypto Analyzer for PEiD*. Accessed: Mar. 26, 2019. [Online]. Available: http://www.dcs.fmph.uniba.sk/zri/6.prednaska/tools/PEiD/plugins/kanal.htm

[24] *Draft Crypto Analyzer* Accessed: Jul. 8, 2019. [Online]. Available: http://www.literatecode.com/draca

[25] L. Auriemma. *Signsrch*. Accessed: Jul. 10, 2019. [Online]. Available: http://aluigi.altervista.org/mytoolz/signsrch.zip

[26] X3chun. *Crypto Searcher*. Accessed: Jul. 8, 2019. [Online]. Available: http://quequero.org/uicwiki/images/Cryptosearcher_2004_05_19.zip

[27] Paradox/AT4RE. *Hash Crypto Detector*. Accessed: Jul. 10, 2019. [Online]. Available: https://github.com/felixgr/kerckhoffs/blob/master/static_tools/HCD.rar

[28] Loki. *SnD Crypto Scanner*. Accessed: Jul. 10, 2019. [Online]. Available: http://www.tuts4you.com/request.php?2222

[29] Guilfanover. *Findcrypt2*. Accessed: Jul. 2, 2019. [Online]. Available: http://www.hexblog.com/?p=28

[30] D. Plohmann. *IDAscope*. Accessed: Jul. 2, 2019. [Online]. Available: https://bitbucket.org/daniel_plohmann/simplifire.idascope/

[31] R. Zhao, D. Gu, J. Li, and H. Liu, "Detecting encryption functions via process emulation and il-based program analysis," in *Proc. 14th Int. Conf. Inf. Commun. Secur. (ICICS)*. Berlin, Germany: Springer, 2012, pp. 252–263.

[32] J. Calvet, J. M. Fernandez, and J.-Y. Marion, "Aligot: Cryptographic function identification in obfuscated binary programs," in *Proc. 19th ACM Conf. Comput. Commun. Secur. (CCS)*, New York, NY, USA, 2012, pp. 169–182.

[33] J. Li, Q. Yin, L. Jiang, and X.-H. Jia, "Analysis of cryptographic algorithms' characters in binary file," in *Proc. 13th IEEE Int. Conf. Parallel Distrib. Comput., Appl. Technol. (PDCAT)*, Dec. 2012, pp. 219–223.

[34] E. C. R. Shin, D. Song, and R. Moazzezi, "Recognizing functions in binaries with neural networks," in *Proc. 24th USENIX Secur. Symp. (USENIX Secur.)*, 2015, pp. 611–626.

[35] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *Proc. 31st Int. Conf. Mach. Learn. (ICML)*, 2014, pp. 1188–1196.

[36] H. Dai, B. Dai, and L. Song, "Discriminative embeddings of latent variable models for structured data," in *Proc. 33rd Int. Conf. Mach. Learn. (ICML)*, 2016, pp. 2702–2711.

[37] L. Massarelli, G. A. Di Luna, F. Petroni, L. Querzoni, and R. Baldoni, "Investigating graph embedding neural networks with unsupervised features extraction for binary analysis," in *Proc. 2nd Workshop Binary Anal. Res. (BAR)*, 2019.

[38] Y. Bengio, R. Ducharme, P. Vincent, and C. Janvin, "A neural probabilistic language model," *J. Mach. Learn. Res.*, vol. 3, pp. 1137–1155, Feb. 2003.

[39] Z. Chen and M. Monperrus, "A literature study of embeddings on source code," 2019, *arXiv:1904.03061*. [Online]. Available: https://arxiv.org/abs/1904.03061

[40] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Proc. Adv. Neural Inf. Process. Syst. (NIPS)*, 2013, pp. 3111–3119.

[41] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," 2013, *arXiv:1301.3781*. [Online]. Available: https://arxiv.org/abs/1301.3781

[42] N. Kalchbrenner, E. Grefenstette, and P. Blunsom, "A convolutional neural network for modelling sentences," 2014, *arXiv:1404.2188*. [Online]. Available: https://arxiv.org/abs/1404.2188

[43] J. Tompson, R. Goroshin, A. Jain, Y. LeCun, and C. Bregler, "Efficient object localization using convolutional networks," in *Proc. 28th IEEE Conf. Comput. Vis. Patt. Recog. (CVPR)*, Jun. 2015, pp. 648–656.

[44] B. Felbo, A. Mislove, A. Søgaard, I. Rahwan, and S. Lehmann, "Using millions of emoji occurrences to learn any-domain representations for detecting sentiment, emotion and sarcasm," 2017, *arXiv:1708.00524*. [Online]. Available: https://arxiv.org/abs/1708.00524

[45] E. Dufourq and B. A. Bassett, "Automated problem identification: Regression vs classification via evolutionary deep networks," in *Proc. South African Inst. Comput. Sci. Inf. Technol. (SAICSIT)*, New York, NY, USA, 2017, p. 12:1–12:9.

[46] *VX Heaven Virus Collection*. Accessed: Jan. 6, 2019. [Online]. Available: http://academictorrents.com/details/34ebe49a48aa532deb9c0dd08a08a017aa04d810

[47] Y. Kim, "Convolutional neural networks for sentence classification," 2014, *arXiv:1408.5882*. [Online]. Available: https://arxiv.org/abs/1408.5882
[48] Gadiluna. *SAFE*. Accessed: May 22, 2019. [Online]. Available: https://github.com/gadiluna/SAFE

LI JIA received the B.Eng. degree from the College of Electronics and Information Engineering, Sichuan University, Chengdu, China, in 2017, where she is currently pursuing the master's degree with the College of Cybersecurity. Her current research interests include binary security, malicious code analysis, and artificial intelligence.

ANMIN ZHOU received the B.Eng. degree from the Northwest Institute of Telecommunication Engineering, Xi'an, China, in 1984. He is currently a Research Fellow with the College of Cybersecurity, Sichuan University, China. His current research interests include malicious detection, network security, system security, and artificial intelligence.

PENG JIA received the Ph.D. degree from Sichuan University, Chengdu, China, in 2019. From 2015 to 2016, he was a Visiting Student with the School of Computer Science, University of Ottawa, Ottawa, ON, Canada. He is currently a Postdoctoral Researcher with the College of Cybersecurity, Sichuan University. His current research interests include binary security, network science, and malware propagation.

LUPING LIU received the M.S. degree from Sichuan University, Chengdu, China, in 2015, where he is currently pursuing the Ph.D. degree with the College of Electronics and Information Engineering. His current research interests include binary security, information extraction, and artificial intelligence.

YAN WANG received the M.S. degree from Sichuan University, Chengdu, China, in 2018, where he is currently pursuing the Ph.D. degree with the College of Cybersecurity. His current research interests include binary security, vulnerability mining, and artificial intelligence.

LIANG LIU received the M.S. degree from Sichuan University, Chengdu, China, in 2010. He is currently an Assistant Professor with the College of Cybersecurity, Sichuan University. His current research interests include malicious detection, network security, system security, and artificial intelligence.