EasyChair Preprint

№ 4345

An Android Malware Detection Method Based on CNN Mixed-Data Model

Andrii Nicheporuk, Oleg Savenko, Anastasiia Nicheporuk and Yuriy Nicheporuk

October 10, 2020

An Android Malware Detection Method Based on CNN Mixed-data Model

Andrii Nicheporuk[0000-0002-7230-9475], Oleg Savenko[0000-0002-4104-745X], Anastasiia Nicheporuk and Yuriy Nicheporuk

Khmelnitsky National University, Khmelnitsky, Ukraine
andrey.nicheporuk@gmail.com,
savenko_oleg_st@ukr.net,eldess06@gmail.com,
yuranichipor2015@gmail.com

Abstract. The paper proposes an Android malware detection method based on convolutional neural network mixed-data model. This data are presented by API method calls and a set of permissions for the Android app. Word2vec technology was used to represent API calls in a vector space, which creates semantically similar feature vectors for related API calls. To represent a set of permissions, each unique permission is encoded as a binary feature that determines the presence / absence of permission in the input sequence. Obtained sequence is then broken down into nibbles and the code "8421" is applied with further normalization of the result. Both types of vectorized data are the inputs to the convolutional neural network. The architecture of the proposed neural network consists of two separate parallel convolutional branches, each of which processes its own type of data, and the fully connected layers. The structure of both branches is the same, which involves placing in each branch two consecutive layers of convolution, where the first layer maps the simple features that will be used by the second layer to represent higher level behavioral patterns. After the convolution layers, there is a pooling layer placed to reduce the dimension of the data. The outputs from both branches of the network are combined to form the input for fully connected layers, which determine the probabilities of belonging suspicious app to one of the classes – malware or benign.

Keywords: Android malware, API calls, permissions, convolution neural network, word2vec.

1 Introduction

In the last decade, humanity has made a significant leap forward in the development of the information technology industry and, in particular, mobile devices operated by the Android operating system. Mobile features have evolved from just making a phone call to supporting the functionality of a full-fledged computer system. However, the advent of new mobile devices features has automatically created new vulnerabilities for them and has driven an increase in the amount of malware, which are trying to use them. This allows malware to perform a full range of malicious activity,

from accessing and stealing private information to displaying undesirable advertisement and spying of the users' activities. Even today, the Google Play web service is not fully protected against the intrusion of malicious software that becomes available for download to Android users. According to the Kaspersky Lab, over 100 million users downloaded the CamScanner text recognition application in mid-2019, which concealed a Trojan program identified as Trojan-Dropper.AndroidOS.Necro.n [1]. There are different reasons for hackers to commit malicious acts. Most often it is obtaining an award, self-affirmation, entertaining, or as a kind of weapon [2]. Therefore, developing new methods for detecting Android malware remains an important task.

The main purpose of our study is to increase the accuracy of a malware detection process by developing a method that would be based on the involvement of neural networks. As a result of ours' research a method for detecting malicious Android software based on the use of mixed-data model for convolutional neural network (CNN) have been proposed. Involvement of convolution layers creates an analogy with the human brain, allowing the identification of local features that are subsequently fed to the input of fully connected layers to form a membership degree of an input object to one of the predicted classes. In the field of pattern recognition, such features may be, for example, the presence of inclined lines at a certain angle. Another important advantage is that the weights in the convolution layer are locally connected and move throughout the feature map. This leads to involving much less of a number of weights compared to fully connected neural network architectures.

As an input data for CNN, we used the API method calls and a set of permissions for Android app. Application Program Interface (API) is a set of procedures that represent an intermediate layer for communicating applications between themselves and the Android kernel. In fact, no one high-level action doesn't take place without the participation of API invocation [3]. Thus, by analyzing them, we could represent the behavior of the application through the sequence of API calls. For example, the sequence getDeviceId (), loadLibrary (), sendTextMessage () might be determined as the behavior of receiving and sending information. The detection process may then be defined as a procedure of search the similarity of the program's behavior with the knowledge about the typical malware behaviors.

The process of obtaining program behavior or the sequence of API calls can be done in two ways [4]. The first way is to convert the dex file (bytecode) to Java source code. This approach involves retrieving class files from the incoming dex file, with dex2jar utility [5] for example, and further conversion to java code. However, it should be noted that in this approach, decompiled java files do not conform to the original code (a reverse operation to obtain a dex file from received java files is not possible). Therefore, the behavior of an application derived from decompiled files may not match with the actual behavior of the application, and cannot be fully used as a feature for Android malware detection.

In our study another approach was used for representing the behavior of an application, which involves the process of disassembling a dex file with apktool utility [6]. This process will produce a set of smali files that are a low-level representation of the Android application. It should be noted that a reverse converting to a dex file is possible, which indicates correlation between both operations.

Except API calls, no less important attributes that can enhance behavior representation is a set of app's permissions. The permissions mechanism restricts access to certain components or functionalities of the application. All permissions used by the application are specified in the AndroidManifest.xml file. According to the results of previous studies, the distribution of permissions in malware and benign applications is differed [7, 8]. Thus, knowledge of attracting permissions may indicate a set of potential actions that will need to be granted.

2 Related works

Considerable attention today is being paid to the problem of Android malware detection. Several solutions have been developed using an academic approach, utilizing features such as permissions, API calls, opcodes, strings, metadata or intents.

In [9], authors had proposed an Android malware detection method based on the method-level correlation relationship of application's abstracted API calls. First, for each Android application's the source code was split into separate function methods and kept the abstracted API calls of them to form a set of abstracted API calls transactions. And then, they calculate the confidence of association rules between the abstracted API calls, which forms behavioral semantics to describe an application. Finally, authors combine machine learning methods to identify the different behavioral patterns of malicious and benign apps to build the detection system.

Another static API-based malware detection system, called MaMaDroid, was introduced in [10]. The MaMaDroid abstracts the API calls performed by an app to their class, package, or family, and builds a model from their sequences obtained from the call graph of an app as Markov chain. The sequences of abstracted API calls for the app, with the transition probabilities used as the feature vector to classify the app as either benign or malware using a machine learning classifier.

Except API calls, permissions can also be used to distinguish behavior between malware and benign Android app. In [11] proposed a permission-based malicious code testing tool called APK Auditor. This offers a user client for granting application analysis requests, an independent database to store application features, and a central server to manage the database and user client. APK Auditor calculates a malware score based on the requestedpermissions and then calculates the malware threshold limit dynamically using logistic regression. Finally, APK Auditor classifies the application as malicious if the calculated application malware score exceeds the malware threshold limit. During a set of experiments on more than 10,000 Android applications, 88% detection accuracy rate was obtained.

In [8] authors propose DroidVecDeep, an Android malware detection method using deep learning technique. During conducting static analyze the authors first extract features such as permissions, actions, sensitive API calls, and use random forests for feature selection. Next they model the extracted features as a document, and then use word2vec to analyze the documents and transform the features into K-dimensional word vectors. And finally the Deep Belief Networks model had been used to establish an optimal detection classifier for Android application classification

However, the main drawback of using only Android permissions as features is a possibility obtain high false positive rate, that is a benign application can mistakenly classified as malicious due to very small difference in permissions sets.

The use of opcodes and residual network as a model for malware detection is presented in [12]. Firstly, the model extracts the opcode sequences using the disassembler. To improve the vector's expressibility of opcodes, Word2Vec strategy was used in the representation of opcodes, and word vector representations of opcodes were also optimized in the process of training iteration. To reduce the redundancy of information, a method of downsampling to organize opcode sequences into opcode matrix was adopted. In order to improve the classification ability of the model, a classifier with more layers and cross-layer connection was proposed to match malicious code in more dimensions based on ResNet.

Another possible direction to detect Android malware is a traffic monitoring. In particular technique for the mobile malware detection based on the malware's network features analysis is proposed in [13]. In the process of monitoring there were collected groups of features, those include the storage resource consumption features, CPU resource consumption features, memory resource consumption features and DNS-based features. As the inference engine for malware detection the support vector machine was used. But despite on the high overall accuracy of technique, it should be noted that not all possible feature vectors, that describe different malware classes, are adequately represented in the training set.

Hybrid technique that involves machine learning is presented in [14]. Authors extract permission, intent, uses-feature, application and API as the static features, and choose the CPU consumption, the battery consumption, the number of running processes and the number of short message as the dynamic features. The raw features were sent to the feature selection module to select some key features and reduce the redundant features based on PCA-RELIEF. Finally, they build a classification model by using SVM and evaluate the unknown Android application by classifying it into malware or benign.

In [15] propose MalDozer, an automatic Android malware detection and family attribution framework that relies on sequences classification using deep learning techniques. Starting from the raw sequence of the app's API method calls, MalDozer automatically extracts and learns the malicious and the benign patterns from the actual samples to detect Android malware. MalDozer can serve as a ubiquitous malware detection system that is not only deployed on servers, but also on mobile and even IoT devices. Malware attribution and detection task is made by using convolution neural network.

3 An Android malware detection method based on CNN mixed-data model

The proposed method for detecting Android malware based on the use of mixed data for CNN consists of two main steps: creating or training a CNN model and applying or deploying the model to detect Android malware.

The training phase involves the creation of a CNN model on a set of training data and involves three sequential steps: preprocessing, vectorization, and directly CNN training.

Preprocessing stage involves obtaining API calls and a set of permissions. The first data are extracted from multiple smali files while the permissions are extracted from AndroidManifest.xml.

The vectorization process uses word2vec technology to represent API calls in vector space. To vectorize a set of permissions, each unique permission is encoded as a binary feature that determines the presence / absence of permission in the input sequence. The sequence is then broken down into nibbles and the code "8421" is applied with further normalization of the result. Both types of vectorized data are inputs to the convolutional neural network.

Training of the convolutional neural network involves sequentially viewing the entire set of training data presented in vector form and generating for each input object a generalization in the likelihood of its belonging to one of two classes. The neural network architecture consists of two separate parallel branches, each of which processes its own type of data, and the fully connected layers. The structure of both branches is the same, which involves placing in each branch two consecutive layers of convolution, where the first layer maps the simple features that will be used by the second layer to represent higher level behavioral patterns. After the convolution layers, there is a pooling layer placed to reduce the dimension of the data. The outputs from both branches of the network are combined to form the input data for fully connected layers.

The deployment phase involves preprocessing for a suspicious Android application, vectorization of its API calls and permissions, and classification using created a neural network model.

Fig. 1 presents the generalized structure of an Android malware detection method based on CNN mixed-data model. Let us take a closer look at each step of the method.

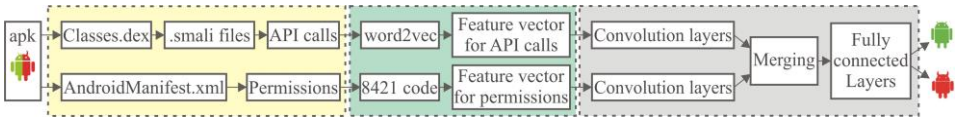


Fig. 1. Generalized structure of an Android malware detection method based on CNN mixed-data model

3.1 Preprocessing

The first step of the proposed method is to extract a list of API calls from smali files and the set of permissions specified in the AndroidManifest.xml file.

Extracting API calls. To get a list of API calls, the dex file is decompiled using the apktool utility. This process will return a set of smali files, containing Android app representation in form of readable Dalvik opcodes. Next stage require parse each smali file and extract the API functions. It should be noted that in the presented method only standard API calls are used, while user methods are ignored. Rejecting custom API calls is explained by the fact that malware can use a large number of third-party methods,

such as some preparatory actions, but standard calls are used to implement the basic functionality (sending a message, checking the network connection, etc.).

The main purpose of getting an API call list is their representation as the behavior of the application. Therefore, in the process of obtaining API calls, it is important to follow their order, which will allows to consider internal relationships between API calls. Because Android applications do not have a single entry point, such as Entry Point in Portable Executable, and applications can be launched not only in the traditional manner by clicking the icon, but in response to intent registered in AndroidManifest.xml, then to collect the behavior of the application, it is important to keep track of all the ways of the application running. To this end, all the services and activities contained in AndroidManifest.xml are added to the list of starting points for collecting application behavior. Next from every starting point, all the API calls that belong to the android, java and javax libraries, that is, calls that interact with the base operating system and defined in the Android specification, are added to the resulting list. If there is a third-party method call, the name of the smali file in which it is implemented is appended to the list of starting points. The algorithm for collecting API calls is shown in fig. 2.

```
Input:  AndroidManifest.xml Manifest
        SetOfSmaliFiles S_i
Output: API_list API
begin
    PointsToBeProcessed=emptyList();
    API = emptyList();
    foreach M_i in Manifest do
        if M_i ∈ AndroidManifest.Activities || M_i ∈ AndroidManifest.Services
            PointsToBeProcessed ← M_i
    foreach S_i in PointsToBeProcessed do
        foreach Instruction in S_i do
            if Instruction ∈ (invoke-* = android, java, javax)
                API ← Instruction
            if Instruction ∈ ReferencesToCustomMethodCall
                PointsToBeProcessed ← Instruction
    return API
end
```

Fig. 2. The algorithm for collecting API calls from *smali* files

Extracting permissions. The list of permissions is obtained by simply parsing the AndroidManifest.xml file. The parsing process considers all sections starting with <uses-permission android… />.

3.2 Vectorization of API calls and permissions

Vectorization of API calls. As result of parsing process will be obtain the list of API calls for each Android application $A = \{a_1,...,a_k\}$. The next step is the vectorization process, that is, the presentation of API calls in the form of real numbers. One way to

achieve this is to encode each API call as a one-hot vector. With this encoding method, the length of the features vector that presents each individual API call, is equal to the dimension of all API calls presented in the dictionary. All number positions are encoded by 0 and the position corresponding to API call is 1. Obviously, with the increasing vector dimension, the amount of the number positions with a zero value increases rapidly. In addition, any two vectors do not correlate in any way among thyself. This situation adversely affects the use of such representation as input for artificial intelligence methods.

In order to obtain a compact vector representation of API calls, the word2vec method [16] was used. This method is used to create embedding predictions in natural language processing systems. Word2vec collects statistics of word co-occurrence in sentences, and then uses neural network methods to solve the task of finding the target word in the context of words (skip-gram model) or finding context in the given word (Continuous Bag of Words – CBoW model). Both models make it possible to maps semantically similar words in close vectors, while distant words in vector space will look different. The classic use of the Word2vec method yields a vector at the output that containing the probability of each word in the dictionary being "in context" for the given word (for the CBoW model). In our study, the word2vec method was used to represent API calls in the form of feature vectors by utilizing the CBoW model. Let's take a closer look at this process.

Suppose an input corpus with size n is specified as $C = \{C_1,...,C_n\}$, where every element of which $C_i \in \{M_i \vee B_i\}$ defines the sequence of API calls of a benign application $B_i = \{a_1^b,...,a_n^b\}$ or malicious software $M_i = \{a_1^m,...,a_n^m\}$. Let every unique API call in corpus C is included in dictionary with size V. Then, represent each API call in form of a one-hot vector such as $r^{a_i} = \{r_1^{a_i},...,r_V^{a_i}\}$, where some $r_l^{a_i} = 1$ and the rest $r_{l'}^{a_i} = 0$, where $l \neq l'$.

Let's denote the context φ for the API call $a_i \in V$ in corpus C as the sequence of API calls that are in the interval $[i-s,..., i-1, i+1,...,i+s]$, where s – is window size. In this case, the context defines the interval which determines the relationship API call a_i with its s-nearest API calls in training corpus. To generate N dimensional feature vectors, a neural network was used that had comprised of input layer neurons $x_1,...,x_s$, where each value is $x_i = \{x_1,...,x_V\}$, one hidden layer $h_1,...,h_N$ and the output layer $y_1,...,y_V$.

Then, given the context, the training set for the neural network in the form of a matrix was formed, each element of which is a one-hot vector:

$$M_{training} = \begin{bmatrix} r^{a_{i-s}} & \dots & r^{a_{i-1}} & r^{a_{i+1}} & \dots & r^{a_{i+s}} & r_1^{a_i} \\ r^{a_{i-s}} & \dots & r^{a_{i-1}} & r^{a_{i+1}} & \dots & r^{a_{i+s}} & r_2^{a_i} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ r^{a_{i-s}} & \dots & r^{a_{i-1}} & r^{a_{i+1}} & \dots & r^{a_{i+s}} & r_P^{a_i} \end{bmatrix} \quad (1)$$

The vectors $[r^{a_{i-s}},...,r^{a_{i-1}},r^{a_{i+1}},...,r^{a_{i+s}}]$ are feed to the input layer of the neural network $x_1,...,x_s$ that $x_j = r^{a_k}$, where $j = \overline{1,s}$, $k = \overline{i-s,i+s}$. The input layer of the neural network is connected to the hidden layer by a weight matrix W of $V \times N$ dimension, and the hidden layer is combined with the output layer by a matrix W' of size $N \times V$.

With the purpose of obtaining embedding for the target API call by context f at first the output of the hidden layer h (2) was calculated. This involves computation of the average value of the k-rows of the matrix W, which were activated by each group of input neurons x_j (i.e. the average for all ARIs calls that represent context s for a_i).

$$h = \frac{1}{s} W \cdot (\sum_{i=1}^{s} x_j) \tag{2}$$

Next the computation of the input for each neuron of the output layer was performed:

$$u_g = {v'_{w_g}}^T \cdot h, \tag{3}$$

where v'_{w_g} is g-th row for output matrix W'.

The last step is to calculate the value of the output layer. The output y_g is obtained by calculating the soft-max function for the input u_g:

$$y_g = p(w_{y_g} \mid w_1,...,w_s) = \frac{\exp(u_g)}{\sum_{g=1}^{V} u'_g} \tag{4}$$

Next, to obtain the best vector representation for API calls, the weights of the matrices are adjusted W and W' the backpropagation method is used.

Thus, as a result of the word2vec method, we will been consider the weighting matrix W as the feature vector for API calls representation.

Vectorization of permissions. Unlike the incoming list of API calls, in which the order of the API calls sets the context for the API call a_i, significance of information about order of permissions location is not high. So let's consider a way of representing Android permissions which based on the encoding of permission depending on its presence or absence in the input sequence.

Therefore, to represent the set of Android permissions $\mathrm{P} = \{\rho_1, \rho_2,..., \rho_t\}, t \le U$ in vector form, let's encode them as bit sequence $Z = \{z_1, z_2,..., z_U\}$, where U – a dictionary containing all available permissions. Then each bit z_i is set to 1, if for ρ_j permission $j = i$, otherwise 0. The result will be a bit sequence with size U, which consists of zeros and ones responsible for the presence or absence of the corresponding permission in P. Next, we have grouped the bit sequence into nibbles and apply-

ing the 8421 code to each nibble was used. Then the bit sequence have been convert to a encoded bit sequence Z_c with size $D_p = U/4$, in which each value is an integer value from 0 to 15. The resulting encoded bit sequence was defined as a vector representation of the set of permissions. In order to process the obtained feature vector by neural network the normalization of its values to real numbers in the range from 0 to 1 with using min-max normalization was done:

In addition, seven critical permissions, most commonly seen in malware, were selected to distinguish more clearly between malware and benign applications. For the ranking of critical permissions (not to be confused with the category of dangerous permissions defined in the Android specification), the previous studies [17, 18] have been considered and our own research has been conducted. As a result we have selected the most commonly used permissions in Android malware: CHANGE_WIFI_STATE, WRITE_SMS, READ_CONTACTS, ACCESS_NETWORK_STATE, BLUETOOTH, INTERNET and READ_PHONE_STATE.

Each of the selected critical permissions is encoded in one bit and does not form as the result of four-bit encoding. Such manner allows to give more weight for the most critical permissions. Thus, the resulting feature vector for permissions consist of encoded bit sequence and seven more bits, each of which represent one critical permission. The process of obtaining feature vector for an Android app's permission is presented in fig. 3.

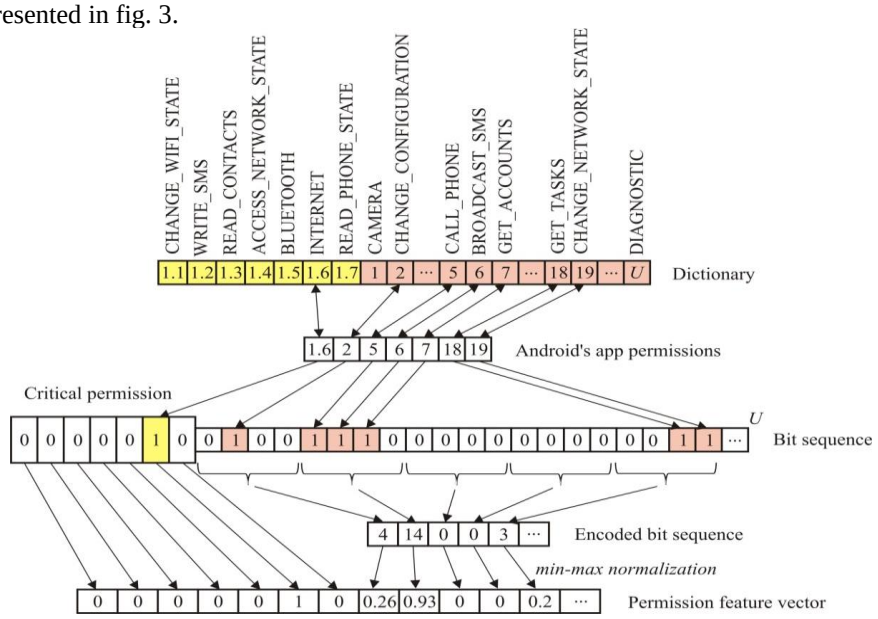


Fig. 3. The process of obtaining feature vector for an Android app's permission

3.3 Convolutional neural network

In order to create a model that will produce a conclusion about the suspicious program's behavior, the neural network is trained. The architecture of the proposed con-

volutional neural network, which uses knowledge about API method calls and set of permissions from the Android application to form a conclusion, is presented in Fig. 4.

The proposed neural network consists of two separate parallel convolutional branches, each of which processes its own type of data (API calls or permissions). As a result of convolution and max-pooling operations, the input data, i.e. behavioral patterns of Android app, is prepared for fully connected layers (FCL). It should be noted that the outputs from both sub-branch are merged, creating the first of three FCL. In order to produce nonlinear decision making, there is one hidden layer between the first and third FCL. The result is provided by the last layer consisting of two neurons.

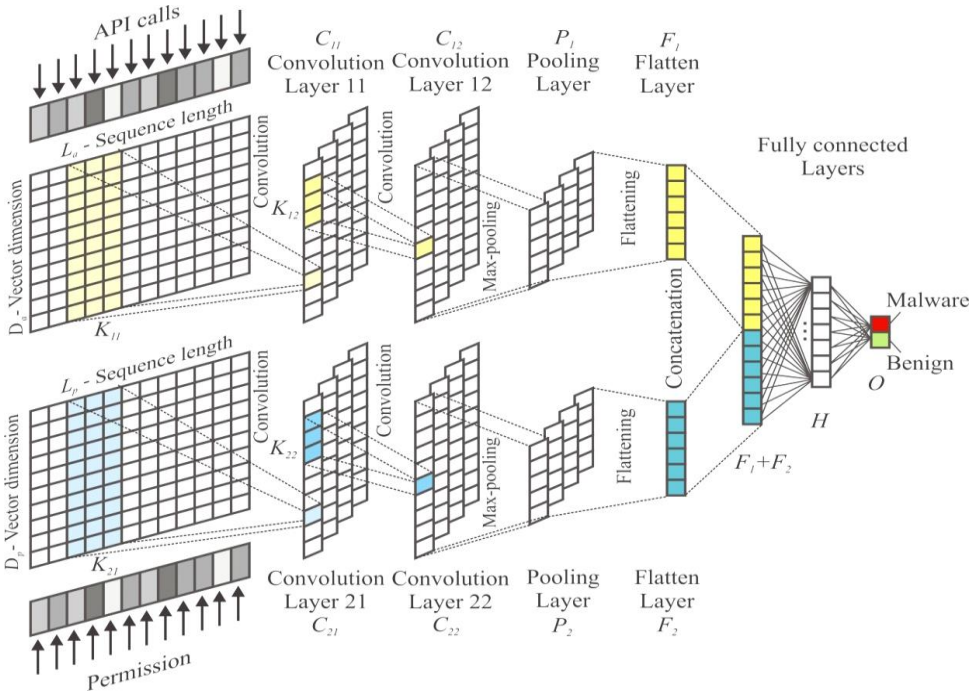


Fig. 4. The CNN Architecture for Android malware detecting

The proposed neural network architecture utilized an approach without convolutional and max-pooling layers alternating. This is due to the fact that after the next pair of CONV + POOL layers, the dimension of the data decreases, which leads to the loss of some information about the input object [19, 20]. In the proposed architecture, in each of the two sub-branch, the two convolution layers C_{11} and C_{12}, as well C_{21} and C_{22}, are placed one after the other, where the first convolution layers C_{11} and C_{21} highlight simple features that will be used by the layers C_{12} and C_{22} to represent higher-level behavior patterns. The input matrices with size $K_{11} \times D_a$ and

$K_{21} \times D_p$, respectively, are feed to the convolutional layers C_{11} and C_{21}. For layers C_{12} and C_{22}, the size of the convolution kernel is $K_{12} \times 1$ and $K_{22} \times 1$, respectively. Following each pairs of convolution layers there is placed one aggregation layer, which reduce the dimension of each type of feature. In order to transform the data into a one-dimensional vector, each sub-branch uses a Flatten layer. After concatenation the data of both sub-branches, the resulting vector with size $F_1 + F_2$ is feeds to FCL. The output layer consists of two neurons that accumulate the probability that a suspicious Android application belongs to one of two classes - malware or benign.

4 Experiments and evaluation

For the purpose of verifying the effectiveness of the proposed method, a number of experiments were conducted. The first experiment involved selecting the optimal parameters for a convolutional neural network. We have considered several templates for neural network and train each of them. This process allowed us to choose the optimal network configuration. After defining the optimal neural network configuration we again have built the model and second experiment to verifying the effectiveness of Android malware detection have conducted.

A set of apk files that corresponded to malicious software or benign applications were selected as test data to evaluate the effectiveness of the proposed method. Malicious samples were obtained from the AndroZoo collection [21] while benign apps from [22]. There are 9198 malware and 7780 benign samples were used for the experiments. Thus, the total number of test data in training corpus C was 16978.

For each sample from the training corpus, the procedure of extracting API calls and a set of permissions was conducted. In order to obtain API calls list, at first apk files was decompiled and a set of smali files are obtained. Next, using a script in Python, an algorithm for extracting API calls from smali files was implemented. The set of Android permissions was obtained by simply parsing AndroidManifest.xml.

In order to create a vector representation of API calls the word2vec method was used, which involved training the neural network to obtain weights between the input and the hidden layer of neurons. The values for the hyperparameters for word2vec were as follows: dictionary for the API calls V =1184 (i.e. the number of unique API calls observable in training corpus C), the size of the context window s for obtaining embedding of API calls was 6, the dimension of the feature vector for API calls (i.e. the dimension of input vectors for CNN) was 64.

To represent each permission in form of the feature vector we have selected the following options: dictionary for permissions U = 168 (the number of unique permissions observable in training corpus C), the dimension D_p of the feature vector for permissions was 49 ($D_p = U / 4 + 7$).

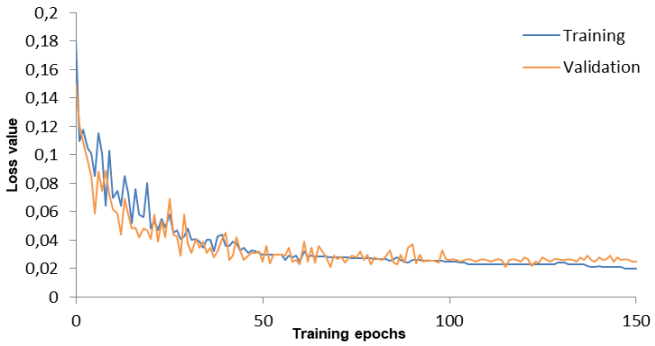
We also unify the length of the API call sequence and set of permissions for each sample from the training corpus. If the application has fewer features than the sequence length L_a or L_p , then the sequence of API calls or permissions of the app

was padded with zeros. then we pad a sequence of API calls or permissions with zeros. If the sequence length for API calls or set of permissions is greater than the value L_a or L_p, then the first L_a or L_p permissions or API calls values were selected from the given sequence.

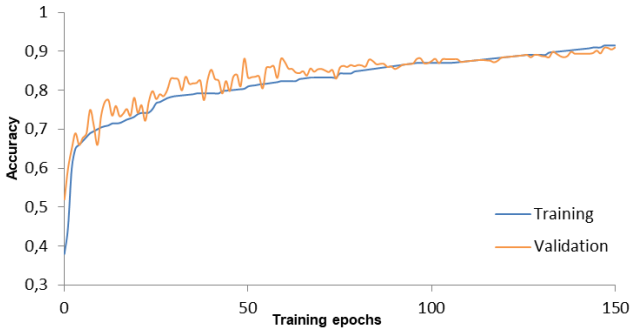
The first experiment involved selecting the most optimal parameters for a convolutional neural network. (fig. 4). To this end, the entire training corpus was divided into two groups: the training set 80% and the data for testing (validation) of the model 20%. The initial value of the number of epochs for training the network was 180 epochs. However, if during the network training, the performance did not change for 10 epochs, the training was terminated. To determine the best configuration of CNN eight templates of configuration options were considered (Table 1). Also for all templates the sequences length L_a and L_p are set to 200 and 50 respectively, and kernel size, padding, stride for $K_{11}, K_{12}, K_{21}, K_{22}$, are set to {3, 1, 1}, {3, 1, 2},{2, 1, 1},{3, 1, 2} respectively. Each neural network was trained and validated. Based on the values of the loss function and accuracy, the optimal values of the network parameters were selected. Neural network training was stopped for 150 epochs, with a loss function of 0.01954. The accuracy value was 0.915. Thus, a set of parameters in template 1 was selected as the optimal neural network configuration. The training and validation process for optimal network configuration (template 1) are shown in Fig. 5.

Table 1. Sets of templates to determine the optimal configuration of the proposed convolution network

№	K_{11}	K_{12}	C_{11}	C_{12}	F_1	K_{21}	K_{22}	C_{21}	C_{22}	F_2
1.	*3*	*3*	*64*	*128*	*64*	*2*	*3*	*64*	*128*	*64*
2.	3	3	64	128	64	3	3	64	128	64
3.	2	2	64	128	64	2	2	64	128	64
4.	3	2	128	64	64	2	3	128	64	64
5.	3	3	128	64	64	3	3	128	64	64
6.	3	3	256	128	64	3	3	256	128	64
7.	4	4	256	128	64	4	4	256	128	64
8.	5	5	128	128	64	5	5	128	128	64



a)



b)

Fig. 5. The Process of neural network training and validation with optimal configuration: loss function (a); the value of accuracy (b)

The second experiment provided verifying the effectiveness of Android malware detection based on the model of the proposed convolutional neural network. The neural network weights were initialized with normal distribution. For all layers, except last, the ReLu activation function was selected. The neurons of the last layer were activated by a softmax function that simulates the probabilities of belonging suspicious app to one of the two classes. The neural network minimized the cross-entropy loss function. In order to reduce the impact of overfitting of the neural network between fully connected layers dropout regularization was used with parameter p = 0.5 (during testing, the dropout parameter was p = 1.0). The learning rate and the batches size were set at 0.001 and 64, respectively. Keras neural network library was used to implement the proposed network [23].

A 10-fold cross-validation was used to evaluate performance metrics. A 10-fold cross-check was used to calculate performance metrics. To this end, 90% of the entire set of C test data was used for model training and 10% for testing. This data selection procedure was performed ten times, each time selecting a different sequence for training and testing. This is allows us to simulate a zero-day malware detection situation. The overall performance of the method was defined as the average of the performance indicators at each of the ten testing stages. After each stage of testing, the values of accuracy, precision, recall and Fscore were calculated as:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}, \tag{5}$$

$$precision = \frac{TP}{TP + FP}, \tag{6}$$

$$recall = \frac{TP}{TP + FN}, \tag{7}$$

$$Fscore = 2 \cdot \frac{precision \cdot recall}{precision + recall}, \tag{8}$$

where, *TP* – number of correctly detected malware, *FN* – the number of malware, wrongly classified as the benign programs, *FP* - the number of benign programs that were wrongly classified as the metamorphic viruses, *TN* – the number of benign programs that were correctly classified.

The results of evaluation of the proposed method effectiveness for detecting Android malware are shown in Table 2. During conducting of experiments the maximum value of detection accuracy was at 0.9412, while the minimum value was 0.9235.The average accuracy was observed at 0.9332 (which is almost the same as the Fscore), while the false positive rate was 3.3%.

Table 2. The results of evaluation of the proposed method effectiveness

№	Observations				Metrics			
	TP	*FP*	*TN*	*FN*	*Precision*	*Recall*	*Fscore*	*Accuracy*
1	8679	519	7283	497	0,9436	0,9458	0,9447	0,9402
2	8636	562	7268	512	0,9389	0,9440	0,9415	0,9367
3	8605	593	7259	521	0,9355	0,9429	0,9392	0,9344
4	8696	502	7262	518	0,9454	0,9438	0,9446	0,9399
5	8055	624	7236	544	0,9281	0,9367	0,9324	0,9290
6	8586	612	7093	687	0,9335	0,9259	0,9297	0,9235
7	8687	511	7249	531	0,9444	0,9424	0,9434	0,9386
8	8624	574	7078	702	0,9376	0,9247	0,9311	0,9248
9	8683	515	7296	484	0,9440	0,9472	0,9456	0,9412
10	8586	612	7093	687	0,9335	0,9259	0,9297	0,9235
Average	**8584**	**562**	**7212**	**568**	**0,9385**	**0,9379**	**0,9382**	**0,9332**

5 Discussion and Future work

During developing the Android malware detection method, it was very important to design and implement not only the neural network architecture, but also to determine the values of hyperparameters that directly have impacting to computational complexity and detection accuracy. In the process of choosing hyperparameters, we had primarily guided by the principle of balancing detection accuracy and computational complexity. Of course, detection accuracy could be improved by increasing the number of consecutive convolution layers and the size of the input feature vectors. However, in this case, the computational complexity of detection method would increase significantly, which would not allow been it's used in real-time malware detection.

In addition, as a disadvantage, it can be noted that the presented method uses static analysis, which do not capable to detect obfuscated malware [24, 25]. To address this shortcoming and as a future study, we will adapt the convolutional neural network architecture to the data that will be obtained during dynamic monitoring of the behavior of the Android application. We are convinced that such combination of data will increase the detection efficiency of malicious Android applications.

6 Conclusion

The paper presents a method of detection Android malware with using a convolutional neural network. The proposed neural network is based on the principle of using mixed data, which represent the knowledge about API method calls and a set of permissions from the Android application. Word2vec technology was used to represent API calls in a vector space, which creates semantically similar feature vectors for related API calls. To represent a set of permissions, each unique permission is encoded as a binary feature that determines the presence or absence of permission in the input sequence. Obtained sequence is then broken down into nibbles and the code "8421" is applied with further normalization of the result. Both types of vectorized data are the inputs to the convolutional neural network.

The architecture of the proposed neural network consists of two separate parallel convolutional branches, each of which processes its own type of data. The outputs from both branches of the network are combined to form the input for fully connected layers, which determine the probabilities of belonging suspicious app to one of the classes – malware or benign.

A number of experiments involving 16978 Android applications were conducted to evaluate the effectiveness of the proposed method. According to the results of those experiments the optimal configuration of parameters for the convolutional neural network was selected and, on the basis of it, the metrics accuracy, recall, precision and F1 score had been evaluated. The average accuracy was observed at 93% with 3.3% of false positives.

References

1. Malicious Android app had more than 100 million downloads in Google Play. Available: https://www.kaspersky.com/blog/camscanner-malicious-android-app/28156/
2. McAfee Mobile Threat Report Q1, 2019. Available: https://www.mcafee.com/enterprise/en-us/assets/reports/rp-mobile-threat-report-2019.pdf
3. Savenko, O., Nicheporuk, A., Hurman I., Lysenko, S.: Dynamic Signature-based Malware Detection Technique Based on API Call Tracing. CEUR Workshop, Vol. 2393, pp. 633–643 (2019)
4. Zheng, M., Lee, P.P., Lui, J. C.: ADAM: an automatic and extensible platform to stress test android anti-virus system. In: Proc. of Detection of Intrusions and Mal-ware & Vulnerability Assessment, pp. 82–101 (2012). doi: 10.1007/978-3-642-37300-8_5
5. Dex2jar. Available: https://github.com/pxb1988/dex2jar.
6. Apktool. Available: http://ibotpeaches.github.io/apktool/
7. Sato, R., Chiba, D., Goto, S.: Detecting Android Malware by Analyzing Manifest Files. In: Proc. of the Asia-Pacific Advanced Network 2013, Vol. 36, pp. 23–31 (2013). doi: 10.7125/APAN.36.4
8. Chen, T., Mao, Q., Lv, M., Cheng, H., Li, Y.: DroidVecDeep: Android Malware Detection Based on Word2Vec and Deep Belief Network. KSII Transactions on Internet and Information Systems, Vol. 13, Issue 4, pp. 2180–2197 (2019)

9. Zhang, H., Luo, S., Zhang, Y., Pan, L.: An efficient Android malware detection system based on method-level behavioral semantic analysis. IEEE Access, Vol. 7, pp. 69246–69256 (2019)
10. Onwuzurike, L., Mariconti, E., Andriotis, P., De Cristofaro, E., Ross, G., Stringhini, G.: MaMaDroid: Detecting Android malware by building Markov chains of behavioral models (extended version). ACM Trans. Privacy Secur., Vol. 22 (2019). doi: 10.1145/3313391
11. Talha, K.A., Alper, D.I., Aydin, C.: APK auditor: permission-based Androidmalware detection system. Digital Investigation, Vol. 13, pp. 1–14 (2015). doi: 10.1016/j.diin.2015.01.001
12. Zhang, X., Sun, M., Wang, J., Wang J.: Malware Detection Based on opcode sequences and ResNet. In: Proc. of International Conference on Security with Intelligent Computing and Big-data Services, pp. 489–502 (2018)
13. Lysenko, S., Bobrovnikova, K., Nicheporuk, A., Shchuka R.: SVM-based Technique for Mobile Malware Detection. CEUR Workshop, Vol. 2353, pp. 85–97 (2019)
14. Wen, L., Yu, H.: An Android malware detection system based on machine learning. In: Proc. of the International Conference on Green Energy and Sustainable Development, vol. 1864 (2017). doi: 10.1063/1.4992953
15. Karbab, E.B., Debbabi, M., Derhab, A., Mouheb, D.: Maldozer: Automatic framework for android malware detection using deep learning. Digital Investigation, Vol. 24, pp. 48–59 (2018). doi: 10.1016/j.diin.2018.01.007
16. Mikolov, T., Sutskever, I., Chen, K., Corrado, G., Dean, J.: Distributed representations of words and phrases and their compositionality. In: Proc. of Advances in neural information processing systems, pp. 3111–3119. Curran Associates Inc., Red Hook (2013).
17. Firdaus, A., Anuar, N.B., Karim, A., Faizal, M., Razak, A.: Discovering optimal features using static analysis and a genetic search based method for Android malware detection. Frontiers of Information Technology & Electronic Engineering, Vol. 19, No. 6, pp. 712–736 (2018). doi: 10.1631/FITEE.1601491
18. Wijesekera, P., Baokar, A., Hosseini, A., Egelman, S., Wagner, D., Beznosov, K.: Android permissions remystified: A field study on contextual integrity. In. Proc. of the 24th USENIX Security Symposium, pp. 499–514 (2015)
19. Um, T.T., Pfister, F. M. J., Pichler, D. et al.: Data augmentation of wearable sensor data for Parkinson's disease monitoring using convolutional neural networks. In: Proc. of the 19th ACM International Conference on Multimodal Interaction, pp. 216–220 (2017)
20. Huang, G., Liu, Z., Maaten, L. et al.: Densely connected convolutional networks. In: Proc. of the 30th IEEE Conference on Computer Vision and Pattern Recognition, pp. 2261–2269. Honolulu, IEEE Press (2017). doi: 10.1109/CVPR.2017.243
21. Allix, K., Bissyandé, T.F., Klein, J., Le Traon Y.: AndroZoo: Collecting millions of Android apps for the research community. In: of the 13th International Conference on Mining Software Repositories, pp. 468–471. IEEE Press, Austin (2016)
22. GetJar. Available: https://www.getjar.com/
23. Keras. Available: https://keras.io/
24. Savenko, O., Lysenko, S., Nicheporuk, A., Savenko, B.: Approach for the Unknown Metamorphic Virus Detection. In: 9-th IEEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems. Technology and Applications, pp. 453–458. IEEE Press, Bucharest (2017). doi: 10.1109/IDAACS.2017.8095052
25. Savenko, O., Lysenko, S., Nicheporuk, A., Savenko, B.: Metamorphic Viruses' Detection Technique Based on the Equivalent Functional Block Search. CEUR Workshop, Vol. 1844, pp. 555–569 (2017)