# Applying deep learning techniques for Android malware detection

Evgeny Pavlenko
Peter the Great
St. Petersburg
Polytechnic University, Russia,
St. Petersburg,
29, Politekhnicheskaya ul.,
+7(812)5527632
pavlenko@ibks.spbstu.ru

Gleb Ignatev
Peter the Great
St. Petersburg
Polytechnic University, Russia,
St. Petersburg,
29, Politekhnicheskaya ul.,
+7(812)5527632
glebignatieff@gmail.com

## ABSTRACT

This article explores the use of deep learning for malware identification in the Android operating system. Similar studies are considered and, based on their drawbacks, a self-designed approach is proposed for representing an Android application for a convolutional neural network, which consists in constructing an RGB image, the pixels of which are formed from a sequence of pairs of API calls and protection levels. The results of the experimental evaluation of the proposed approach, which are presented in this paper, demonstrate its high efficiency for solving the problem of identifying malicious Android applications.

## CCS Concepts

**Security and privacy→Systems security→Operating systems security→Mobile platform security**

## Keywords

Information security, Android OS, mobile security, malware, application analysis, deep learning, convolutional neural network, Android application, Android security, malware detection

## I. INTRODUCTION

According to the latest report of various operating systems distribution on the market [1], Android occupies more than 75% of the market and it's significantly ahead of its closest opponents. Currently, Android runs on a large number of different devices and form factors, whether it's a smartphone, tablet, e-book, watch, fitness bracelet, laptop, TV, car computer, smart glasses and many other devices. Given the prevalence, flexibility and versatility of Android OS, it is not surprising that this operating system is also the most popular among malware developers, and Android is not only the most popular platform for malware among mobile systems, but also among desktop, where it is much ahead of its nearest competitor – Windows OS [2].

In the latest report of McAfee antivirus company for the first quarter of 2018 [3] it is stated that the problem of protecting Android users remains relevant, since despite recently launched Google Play Protect in the official Google Play Market, which, according to developers, checks more than 50 million applications everyday using various machine learning algorithms, the growth of malware continues, and the system itself does not cope with tests on typical types of malicious applications conducted by the company.

In order to solve the security problem posed in this paper, we propose an approach for Android malware detection based on deep learning, namely, in this work we use a convolutional neural network (CNN) to detect malicious applications using self-designed representation of an Android application.

The paper is organized in the following manner: Section II reviews previous studies on the subject of this paper; Section III proposes a new approach for identifying malicious Android applications based on a convolutional neural network and self-designed application representation; Section IV describes the process of preparing a dataset for the training and evaluation of a neural network and a model of this network; Section V presents the results of an experimental evaluation of the program implementation of the proposed approach; and finally, Section VI presents the conclusions and possible direction of the further works in this scope.

## II. RELATED WORKS

One of the most significant advantages of deep neural networks in addition to their phenomenal efficiency, superior to the effectiveness of other machine learning algorithms for the same range of tasks, is their ability to generate features on fly based on the input data. Otherwise, to achieve high results, researchers have to come up with sophisticated methods for analyzing Android applications, as the authors of [4] did, who carried out a deep analysis of the application structure and used the theory of matrices, graphs, and machine learning to identify malicious applications. The application of classical machine learning algorithms somewhat simplifies the process of constructing classification criteria, but does not relieve from the laborious process of feature vector construction, moreover, such algorithms significantly limit the researcher in the length of this vector. For example, in [5], researchers use static and dynamic analyzes to build a feature vector, but these features are just the facts of the existence of an action or the use of some API, permission, etc., i.e. there is no way to track any chains of similar facts that could more clearly describe the behavior of an application.

Although deep neural networks have the aforementioned strengths, first researches on exploring the security of Android applications using deep neural networks have not fully exploited these strengths. Therefore, this studies were very much like studies which used classic machine learning algorithms like SVM and Naïve Bayes for classification, the only difference was the classification algorithm. For example, in [6], authors used a deep belief network (DBN) as a classifier, and the Android application was represented by its permissions, Intents, system commands, list of premade suspicious APIs. Thus, only the classification algorithm has changed. In [7], researchers also used a DBN and a large set of manually designed features derived from static and dynamic analysis of the application.

Soon enough the researchers abandoned the idea of manual features design and came to the conclusion that, from the semantic point of view, the most effective choice for analysis

would be the DEX-files containing the compiled code of an Android application. The analysis of this code will allow to use all the strengths of the deep learning.

T. Huang and H.Kao in [8] used a convolutional neural network to identify malicious Android applications which they presented as an RGB image composed of a sequence of DEX-file opcodes – three hexadecimals per pixel. This representation has a number of drawbacks: firstly, the DEX-file contains a lot of elements used by each application – both malicious and benign, so these elements are just an extra set of input data, so they just consume memory and computational resources and are useless for application classification, secondly, the opcode sequence in the DEX-file does not correspond to the actual order of execution, which does not reflect the semantic features of the application.

In addition to direct representation of the DEX-file binary, the disassembled smali bytecode, which is preferred instead of Java due to the simplicity of its syntax and structure, was more actively used as input data. Smali has only 218 different instructions which represent all possible operations and language constructs, moreover, you may have difficulty in decompiling DEX in Java if it was optimized and obfuscated previously.

N. McLaughlin, J. Martinez del Rincon et al. in [9] represented Android application as a sequence of smali instructions opcodes and then they used a convolutional neural network for the Android malware detection. The main disadvantage of this application representation was that all possible instructions were analyzed, but their parameters were discarded, and therefore valuable semantic information was lost.

Authors in [10, 11] concluded that the most valuable information about the behavior of an Android application is its API calls, so application representation should be composed of API calls, and it should not be done as in the early works, for example in [12], where the vector of unique API calls among dataset is used, it should be done using API calls sequence, which is built from application control flow graph. The main advantage of this approach is that, in addition to the presence and absence of API calls, it is possible to track some specific sequences for different classes of applications, which the deep neural network, due to its ability to generate features on fly, can extract and then learn to find them in other applications. The sequence of API calls carries only semantically useful information and almost no "noise" as opposed to previous application representations, where representation contained a lot of unnecessary information. Authors of these papers used convolutional neural network for applications classification as well.

The problem with the approaches listed above is that of the entire set of components of the Android application package they take into account only DEX files. However, it should be taken into account that the Android application permissions defining the functionality of the application also carry valuable semantic information. It is possible to achieve classification accuracy more than 90% based only on permissions [13], so if properly applied, then you can get a significant increase in the accuracy of classification of malicious and benign Android applications.

# III. APPROACH

The approach proposed in this paper is presented in Figure 1 The main idea of the approach is to represent an Android application as an image for later analysis by a convolutional neural network. In this image, the pixels represent the sequence of pairs of API call and the corresponding security level that is derived from the permission that is required for the API call execution.

## A. Representing Android application as a sequence of API calls and security levels

According to the Android SDK, each Android permission is assigned to a protection level, there are three levels: normal, dangerous, signature (there is another level – signature|privileged, but for sake of simplicity we will consider it as the same level as signature). In turn, every API call can be assigned to a permission, if it is required for its execution. Thus, for each call API, you can map the protection level in accordance with the permission(s) required by it. As a result, Android application can be represented by the following sequence of tuples, expressed by the formula 1:

$$\{API_i, PL_j\}_k, \quad (1)$$

where $API_i$ – some API call from the basis, i.e. the set of unique API calls built from the whole Android applications dataset, $PL_j$ – some protection level, and $i: 0 \le i \le \#API$, $j: 0 \le j \le \#PL$, $k: 0 \le k \le L$, where $L$ – the API sequence length of a specific Android application.

In turn the protection level $PL_j$ is expressed as a double mapping of the call API represented by the formula 2:

$$PL_j = toPL(toP(API_i)), \quad (2)$$

where $toP: API \to (P \cup \emptyset)$ – a function that maps API call to the required permission $P$ or to $\emptyset$ (empty set), if permission is not required, and $toPL: (P \cup \emptyset) \to PL$ – a function that maps some security level or an empty set to a protection level.

The use of the term "protection level" in the context of this paper is not equivalent to the same term in the context of the Android SDK. The protection level in the context of this work is a superset of the protection level from the Android SDK. In addition to the levels from SDK, protection levels set $PL$ contains a special level that matches API calls that do not require any permission for its execution, as well as the level that API calls are mapped to if these calls are not from the android.* packages, to which the concept of "permissions" is not applicable. Some set of such APIs will be taken into account in this work. In summary, set of protection levels can be represented as the set of five in formula 3:

$$PL = \begin{Bmatrix} none, unknown, normal, \\ signature, dangerous \end{Bmatrix}, \quad (3)$$

where the first two levels are the two levels described above that extend the notion of "protection level" of the Android SDK, and the last three correspond to the protection levels from the SDK.

Since Google does not provide any way to get corresponding permission for an API call except for official documentation that is unsuitable for automated processing, the PScout [14] and axplorer [15] projects, which are static code analyzers for the Android SDK code, are used to match an API to the required permission. The output of these projects contains literally what we need – API calls to permissions mappings.

## B. Representing a sequence of pairs as an RGB image

With each new version of Android, the number of available APIs is growing, in addition, Java and Javax APIs are taken into account in this work, so the exact number of APIs cannot be named, these are hundreds of thousands of APIs, which

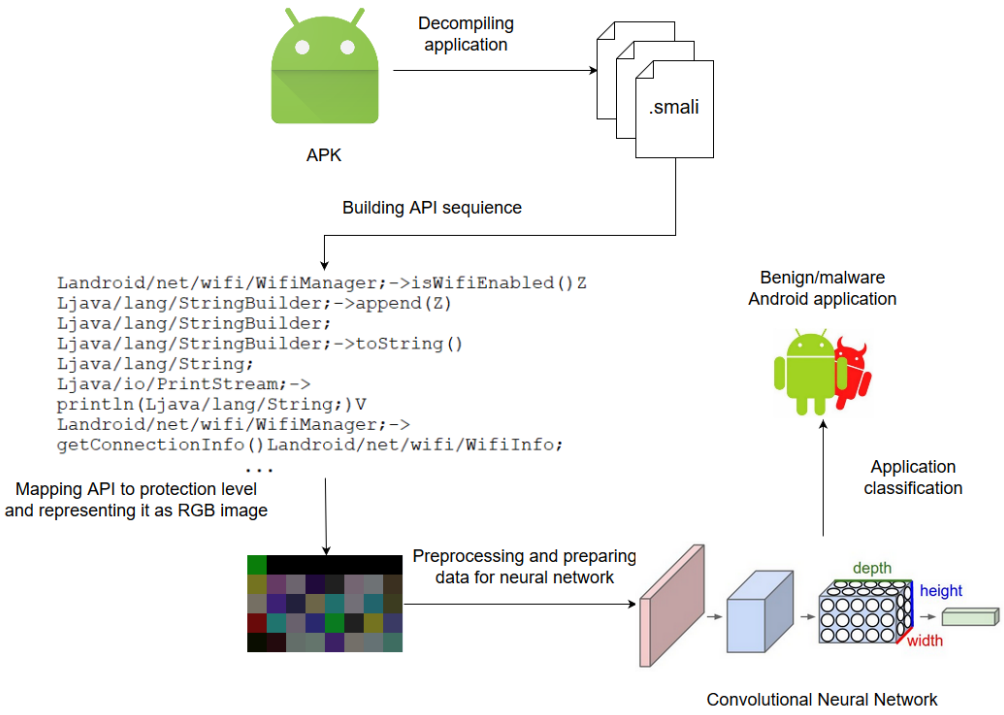


**Figure 1 – Proposed approach**

definitely lies between $2^{16}$ and $2^{24}$. In this connection it is naturally to encode each API call in the form of a three-byte vector or RGB pixel. Recall that for each API call there is a corresponding protection level. Taking into account that all 3 channels of the pixel are already occupied by the API code, for the protection level it would be possible to give the alpha channel and order the level codes in such a way that the more dangerous protection level corresponds to a larger numerical value. Thus, more transparent pixels will correspond to less dangerous APIs, and less transparent ones will be more dangerous.

This representation seems to be successful at first glance, but it has one significant drawback. Imagine, we have an RGB image size of 200x200. Let's calculate the number of input neurons: $200 \cdot 200 \cdot 3 = 120$ thousand neurons. For a similar RGBA image, the number of parameters will increase by one third and become equal to 160 thousand parameters. The increase in number of parameters will be even more significant as image spatial size grows.

Nevertheless, if you slightly modify the idea of representing the pair $\{API, PL\}$ as a RGBA pixel, then it is possible to get rid of the above drawback. To do this, you need to use a conversion from RGBA palette to RGB, which is performed relative to some background, to which the RGB pixel values will tend when the alpha value is increased, i.e. increase of transparency. For example, if you convert from a relatively white background (#ffffff), the more transparent RGBA pixels are converted to lighter RGB pixels, i.e. aspiring to a white background. In this paper, black (# 000000) was selected as the background, so the more transparent pixels will be converted to darker ones. This option is preferable in connection with the presence of max-pooling layers in a convolutional neural network, performing the MAX operation on elements of some region. Thus, pixels with a higher value will be selected, and therefore probably more dangerous APIs.

To create a denser distribution of API values by 3 bytes, the fast non-cryptographic 32-bit hash function called MurmurHash3 [16] is used. After computing the hash value, its least significant 24 bits are taken, resulting in collisions, but as it turned out in practice, the number of collisions is extremely small in comparison with the number of unique API calls, so this problem can be neglected.

Earlier in this paper it was said that the protection level acts as an alpha channel. The alpha channel is represented by a value from 0 to 1, respectively from absolutely transparent to completely opaque. Table 1 shows the protection levels present in this work:

**Table 1 – Protection levels**

| Protection level | Numerical value | Short description |
|---|---|---|
| NONE | 1 | Assigned to the Android API, which has not been assigned any permission |
| UNKNOWN | 2 | Assigned to Java and Javax API calls |
| NORMAL | 3 | Corresponds to the Android SDK permissions group with normal protection level |
| SIGNATURE | 4 | Corresponds to the Android SDK permissions group with signature protection level |
| DANGEROUS | 5 | Corresponds to the Android SDK permissions group with dangerous protection level |

Protection level numerical value mapping into a value range from 0 to 1 is performed elementary according to the following formula 4:

$$PL^{\alpha} = \frac{PL_i}{PL_{dangerous}}, \quad (4)$$

where $PL_i = \{1, \dots, 5\}$, $PL_{dangerous} = 5$, and $PL^{\alpha}$ is a value of $PL_i$ in range from 0 to 1.

An example of the above RGBA-RGB conversion is shown in Figure 2, where $RGBAtoRGB$ represents the above procedure for converting an RGBA pixel to RGB with a black background. As you can see in the figure, RGBA pixels with the same RGB channel values, but different alpha channel values are converted into correspondingly darker and lighter RGB pixels:

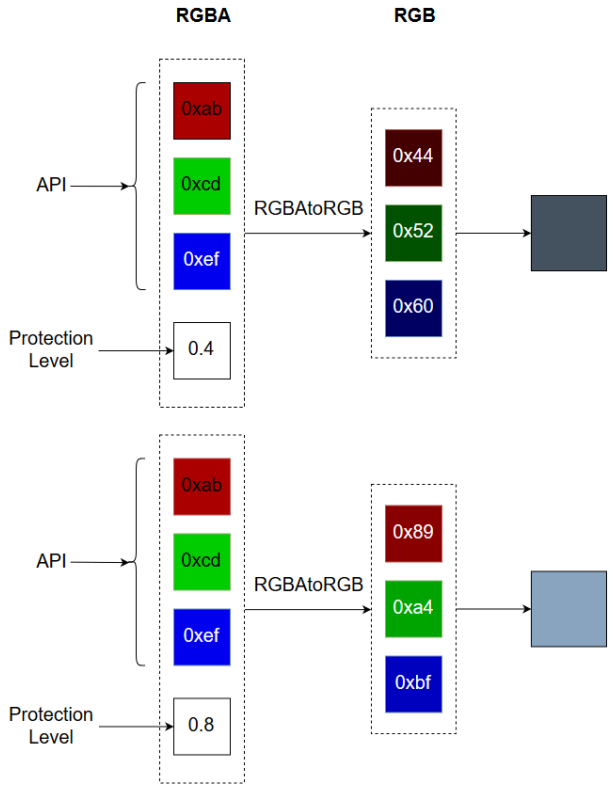


**Figure 2 – – Example of RGBA-RGB conversion**

## C. Convolutional neural network as a classifier

As mentioned earlier, in this paper we use a convolutional neural network to classify input images representing benign and malicious Android applications. A convolutional neural network consists of several different types of layers, each one of them performs a certain function. The choice of types, number and sequence of layers, in other words, the construction of the architecture of a convolutional neural network is a complex practical problem, since its construction is primarily based on data, in addition, the convolutional neural network model has many parameters which are determined in the course of experiments on training a neural network. These parameters are usually called hyperparameters.

The input for the neural network is a some volume, i.e. a three-dimensional tensor of dimension $W \times H \times D$, where $W$ is image width, $H$ is image height and $D$ is image depth or the number of channels. In this work $D = 3$ for the input volume, since input images have 3 channels which are R, G and B. The neural network is trained with the teacher, i.e. together with the input image $X$, representing some Android application, a label $y = \{0,1\}$ is given to neural network, where $y = 1$, if the application is malicious, and $y = 0$, if it is benign. The training of any deep neural network, including convolutional one, is carried out for several epochs, i.e. several runs of the training samples. At the end of each training epoch test samples, on which the neural network was not trained, are used to evaluate the accuracy of neural network. As a result of training, we have a ready-made classifier capable of detecting malicious Android applications with specified accuracy.

# IV. IMPLEMENTATION

## A. Building dataset

To conduct research, a set of malicious and benign applications was collected. To collect 1387 applications from the third-party Android application repository Android Drawer [17], an APK-file crawler was implemented using the Python Scrapy package. All applications were verified with the help of VirusTotal and 12 of them turned out to be malicious. As a result, a total of 1375 benign applications were received. Other 5879 benign Android apps were taken from the PlayDrone APKs set [18]. This set has been used several times by researchers as a sample of benign Android applications [19, 20]. In total, 7254 benign applications were collected.

All 24,553 malicious Android applications were obtained from the Android Malware Dataset project [21]. In this repository APK files are sorted by the corresponding families of malicious applications.

As a result, the application collection consists of 31653 Android applications in total, among which 7214 benign and 24439 malicious.

The whole process of data preparation can be represented using the diagram shown in Figure 3, the colored items in the figure indicate the phases performed by the Androguard package for reverse engineering of Android applications [22].

The API basis is a set of unique API calls for all applications from the collection. To build this basis, you need to extract all the APIs used by each application from the whole collection, and exclude all duplicates from this set. As a result, a basis containing 41415 call APIs was constructed.

After the basis construction, you need to extract the call API sequences for each application. Algorithmically, this step can be written as follows (all variables and functions are specified in terms of the Androguard package):

1. Initialize empty list $api\_sequence$.
2. For each code block $code\_block$:
3. For each instruction $ins$ in the block $code\_block$:
4. Get instruction opcode $opcode \leftarrow ins.get\_op\_value()$;
5. If $0x6e \leq opcode \leq 0x78$:
    a. Get list of operands $operands = ins.get\_operands()$;
    b. Get API call signature $api = operands[-1]$
    c. If $api$ in basis $UniqueAPIs$ and two previous elements of $api\_sequence$ are not equal to $api$:
        i. Append $api$ to $api\_sequence$.

At the 5th step only the opcodes from 0x6e to 0x78 are filtered, the given range of opcodes belongs to the smali instructions of the type invoke- *, which are used to call the methods in various situations. In 5.b. API signature is a method name, its parameter types and return type. In 5.c. of the present algorithm, you can

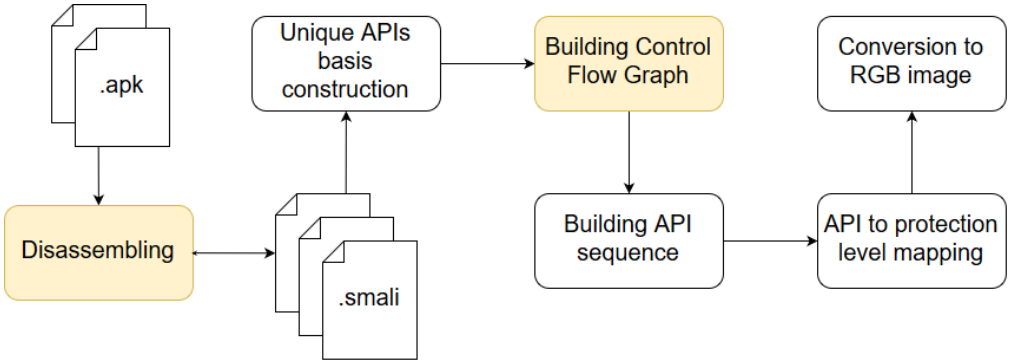


**Figure 3 – Data preparation process for neural network**

see the condition "the two previous elements of *api_sequence* are not equal to *api*". This condition is added in order to exclude long sequences of repetitive API calls that do not carry valuable information, but rather a noisy sequence.

The lengths of the received API call sequences vary greatly: there are sequences that have a length of a couple of dozen API calls, and there are sequences of hundreds of thousands length. It is known that in order to solve the classification problem (in the presence of a fully connected layer at the end), it is necessary that the volume size of the neural network input layer be fixed. Assuming this, the image size was chosen taking into account the maximum size of the sequence with a little window and is equal to 384x384 pixels (it is preferable to choose size which is divisible by 2 many times).

At the last step, the API calls sequence is converted to image RGB pixels according to how it is described in Section III. An example of the resulting image is shown in Figure 4.

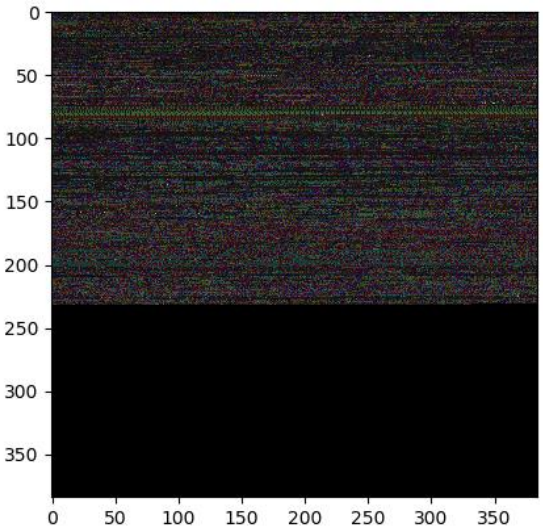


**Figure 4 – An example RGB image corresponding to some Android application**

Algorithmically, this step can be written as follows:

1. Compute the hash function Murmur3Hash from API call signature.
2. Take 24 least significant out of obtained hash value.
3. Use PScout and axplorer to get a list of permissions corresponding to the current API call or empty set if no permissions are required.
4. Using the permission mapping table and its protection level, obtain the maximum protection level among the permissions required to complete the API call.
5. Convert the resulting level to a value in range from 0 to 1, according to formula 3 from Section III.
6. The 3 bytes of hash value and a fraction of the protection level are compiled into RGB pixel and alpha channel respectively and are given to function that convert RGBA pixel to RGB with black background.

The resulting data set is saved as a serialized Python dictionary similar to CIFAR-10 [23].

## *B. Neural network model*

In this paper, as a software package for building a convolutional neural network, we use the Keras package [24], which runs on top of TensorFlow. The experience of various convolutional networks that have won the ImageNet Large Scale Visual Recognition Challenge (OLSVRC) competition of various years shows that the most successful template for building a convolutional network architecture is the following template:

```
INPUT -> [[CONV -> RELU]*N -> POOL?]*M ->
          [FC -> RELU]*K -> FC,
```

where `*` means repetition, a `POOL?` – optional pooling layer. Moreover, `M, N, K >= 0` and usually `N <= 3, K < 3`.

Based on this template, the following architecture of convolutional neural network in Keras terminology was constructed:

```
INPUT -> [CONV -> RELU -> CONV -> RELU ->
          POOL -> DROP]*2 ->
    FLAT -> [FC -> RELU -> DROP] -> FC
```

For the selected model of the convolutional neural network, the following layer hyperparameters were selected, presented in Table 2. The value of the padding parameter, which is equal to 'same', means that a zero padding of such a size is added that the output volume of current layer is spatially, i.e. by width and height, remains unchanged. A padding value of 'valid' means there is no zero padding, and therefore the spatial size of the output layer is decreased.

Taking into account the chosen model of the convolutional neural network, as well as the above-mentioned values of the hyperparameters, we have the following volume dimensions obtained by input volume dimension conversions during its feedforward phase, presented in Table 3.

**Table 2 – Hyperparameter values for every layer**

| Keras model layers | Hyperparameter | Value |
|---|---|---|
| conv2d_1_input | input_shape | (31807, 384, 384, 3) |
| | filters | 32 |
| | kernel_size | (3, 3) |
| | padding | 'same' |
| | strides | (1, 1) |
| conv2d_1 | filters | 32 |
| | kernel_size | (3, 3) |
| | padding | 'valid' |
| | strides | (1, 1) |
| conv2d_2 | filters | 64 |
| | kernel_size | (3, 3) |
| | padding | 'same' |
| | strides | (1, 1) |
| conv2d_3 | filters | 64 |
| | kernel_size | (3, 3) |
| | padding | 'valid' |
| | strides | (1, 1) |
| max_pooling2d_1,2 | pool_size | (2, 2) |
| | strides | (2, 2) |
| dense_1 | units | 256 |
| dense_2 | units | 2 (0, 1) |
| activation_1…5 | activation | 'relu' |
| activation_6 | activation | 'softmax' |
| dropout_1,2 | rate | 0,25 |
| dropout_3 | rate | 0,5 |

**Table 3 – Input volume dimension conversions**

| Layer | Data volume dimensions | | |
|---|---|---|---|
| | Width | Height | Depth |
| Input | 384 | 384 | 3 |
| 2×Conv2D | 384 | 384 | 32 |
| MaxPooling2D | 192 | 192 | 32 |
| Dropout | 192 | 192 | 32 |
| 2×Conv2D | 192 | 192 | 64 |
| MaxPooling2D | 96 | 96 | 64 |
| Dropout | 96 | 96 | 64 |
| Flatten | 1 | 1 | 589824 |
| Dense | 1 | 1 | 256 |
| Dense | 1 | 1 | 2 |

## V. EXPERIMENTAL EVALUATION

To conduct experiments on the selection of hyperparameters, training the neural network and performance evaluation, a virtual machine was used on the cloud service Paperspace [25], which consists of the following components, presented in Table 4:

**Table 4 – Virtual machine components**

| Component | Summary |
|---|---|
| CPU | Intel Broadwell E5 v4 (8-core) |
| RAM | DDR4 32 Gb |
| GPU | Nvidia Quadro P4000 (8 Gb) |
| SSD | 100 Gb |
| OS | Ubuntu 14.04 LTS |

For neural network performance evaluation in this paper we use accuracy metric, presented in formula 5:

$$Accuracy = \frac{TP + TN}{Total}, \quad (5)$$

where $TP$ (true positive) – the number of correctly classified malicious Android applications, $TN$ (true negative) – the same but benign, and $Total$ – total number of test application samples.

As indicated in Section IV, the present dataset is highly imbalanced towards malicious applications, which can result in an increased number of false positives, since the neural network will be better at identification of one class than the other. In this regard, we will conduct two experiments: in the first experiment, we simply train the neural network on this dataset, and in the second experiment we artificially balance the training part of the initial dataset. In both experiments, we will divide the original dataset into training set (the ones on which the neural network learns) and test set (on which the accuracy is estimated) in a ratio of 2 to 1.

The values of the hyperparameters of the learning algorithm used in the experiments are presented in Table 5:

**Table 5 – Learning algorithm hyperparameters**

| Hyperparameter | Value |
|---|---|
| learning_rate | 0.005 |
| loss | categorical_crossentropy |
| epochs | 100 |
| batch_size | 32 |
| decay (SGD) | 1e-6 |
| patience (EarlyStopping) | 5 |

In the first experiment, when dividing the dataset in this ratio, we have 21112 benign applications and 10541 malicious applications. The results of the accuracy evaluation and the loss function curves are shown in Figure 5. According to the figure, a strong overfitting is observed, since the accuracy curve (loss function) of the training set is located below (above) the test set curve. Since one of the main approaches of overfitting reduction is to increase the number of training samples, it is expected that an artificial generation of samples should also positively affect the neural network performance. Nevertheless, as a result of the experiment the accuracy of 92.84% was achieved.

For the second experiment, using the imbalanced-learn Python package [26], the original imbalanced dataset was balanced using a simple algorithm for generating synthetic samples – Random Over-Sampling (ROS), which is a nutshell simply

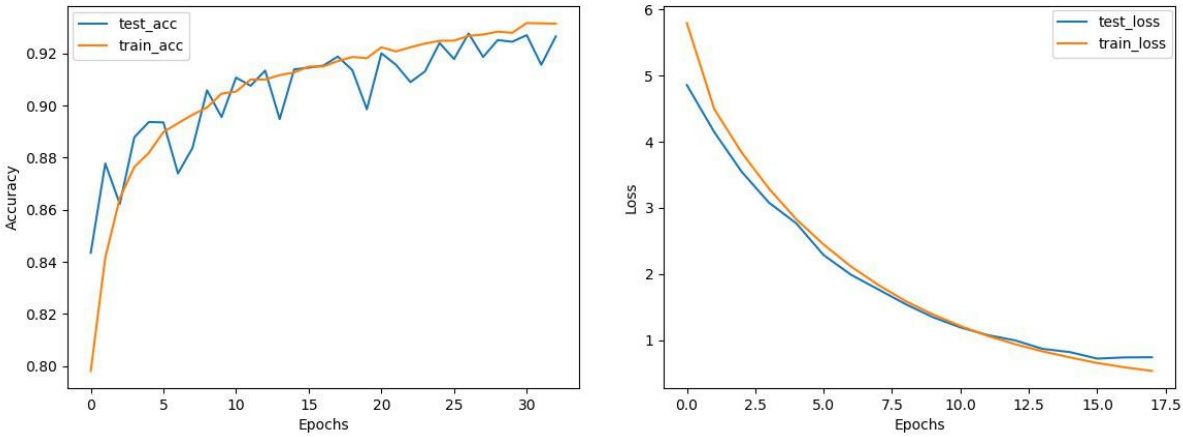


**Figure 5 – Accuracy and loss function curves for experiment #1**

randomly picks samples from already existing ones. In this paper, this algorithm is used due the fact that more efficient algorithms like SMOTE and others are very resource consuming in terms of hardware, and in present study it is impossible to apply them. This is primarily due to the fact that the input data size is very large: $384 \cdot 384 \cdot 3 = 442368$.

For synthetic generation, it is recommended to use only a training set, since if you use the whole dataset, then there may be a correlation between the training samples and the test samples, and then the independence of the test set will be lost, and this is its key feature.

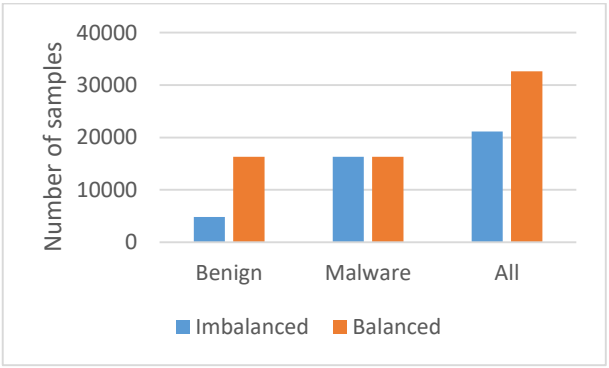


**Figure 6 – Results of ROS operation**

In this experiment, after the initial dataset division, we have all the same 21112 training samples, among them 4810 samples of benign applications and 16302 – of malicious, and 10,541 test samples, where 2,382 and 8,159 samples of benign and malicious applications, respectively. As a result of the operation of the ROS algorithm, the missing number of benign samples of in the training set was completed to the number of samples of malicious applications (Figure 6).

Thus, the size of the balanced training set is 32609, and the total size is 43145 samples. The results of training and performance evaluation of the neural network in this experiment are shown in Figure 7.

As a result of the artificial balancing of the dataset, the accuracy of 93.64% was achieved, i.e. we have got an increase in almost one percent. Thus, increasing the number of samples by means of synthetic sample generation, even with a simple algorithm such as ROS, gives a significant increase in the accuracy of the classifier.

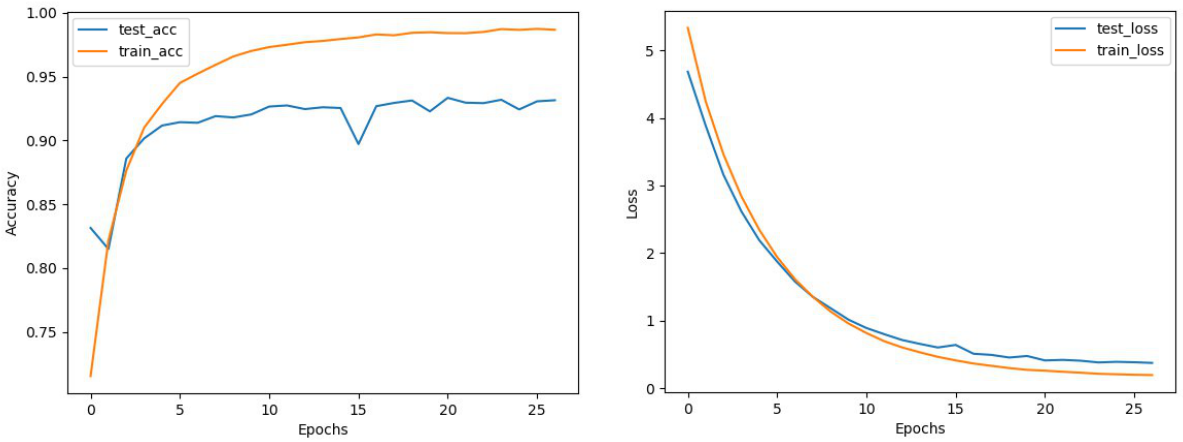


**Figure 7 – Accuracy and loss function curves for experiment #2**

## VI. CONCLUSION

As a result of the work, different approaches of Android application representation for deep neural network for the purpose of identifying malicious software are examined and their drawbacks are revealed. Then self-designed approach of Android application representation is proposed, in which application is represented as an RGB image whose pixels are constructed from a sequence of pairs of API call and protection level, which is derived from permissions that are required to execute this API call. As a result of the experimental evaluation of the implementation of the described approach, it the classification accuracy of 93.64% is achieved due to the synthetic generation of benign application samples, which was carried out in order to reduce the overfitting of the convolutional neural network, and also to balance the dataset used.

In future studies it is supposed to be more reasonable to approach the problem of preprocessing a sequence of pairs (API call, protection level) for further presentation for a deep neural network (the transition to another architecture is possible, for example, to the LSTM network), namely, it is planned to solve the problem of collision of encoded values of various API calls (hash values in this paper), as well as the problem that in the current approach a dangerous API call can be encoded to such a small numeric value that the value of the protection level will not fix the situation of how small value this dangerous API has, and as a result this API call will look like it is of normal protection level or even less. Finally, it is planned to solve the problem of extremely large sequences. Since every year applications size increases, the current approach will have to increase the size of the image every time application of a bigger size come out, which is very inefficient with respect to RAM and disk memory due to the presence of small-size sequences for which the image construction requires the huge zero pixel padding, which significantly increases disk memory occupied by the whole dataset.

## VII. REFERENCES


1. Mobile Operating System Market Share Worldwide, http://gs.statcounter.com/os-market-share/mobile/worldwide.
2. Android vs iOS vs Windows: Which suffers most infections? Nokia reveals all, https://www.zdnet.com/article/android-vs-ios-vs-windows-which-suffers-most-infections-nokia-reveals-all/.
3. McAfee Mobile Threat Report, https://www.mcafee.com/cn/resources/reports/rp-mobile-threat-report-2018.pdf.
4. Zegzhda, P., Zegzhda, D., Pavlenko, E. and Dremov, A., Detecting Android application malicious behaviors based on the analysis of control flows and data flows, Proc. of the 10th Inter. Conf. on Sec. of Inf. and Net., 2017, pp. 280-283.
5. Pavlenko, E.Y., Yarmak, A.V. and Moskvin, D.A., Application of clustering methods for analyzing the security of Android applications, Autom. Control Comput. Sci., 2017, no. 8, pp. 867–873.
6. H. Fereidooni, M. Conti, D. Yao, A. Sperduti, ANASTASIA: ANdroid mAlware detection using STatic analySIs of Applications, 2016 8th IFIP International Conference on New Technologies, Mobility and Security (NTMS), 2016.
7. Z. Yuan, Y. Lu, Y. Xue, Droiddetector: android malware characterization and detection using deep learning, Tsinghua Science and Technology, 2016. – Vol. 21. – pp. 114-123.
8. T. Huang, H. Kao, R2-D2: ColoR-inspired Convolutional NeuRal Network (CNN)-based AndroiD Malware Detections, arXiv.org e-Print archive, https://arxiv.org/pdf/1705.04448.pdf, 2017.
9. N. McLaughlin, J. Martinez del Rincon, B. Kang, S. Yerima, P. Miller, S. Sezer, ... G. Joon Ahn, Deep Android Malware Detection, In Proceedings of the ACM Conference on Data and Applications Security and Privacy (CODASPY) 2017 Association for Computing Machinery (ACM), pp. 301-308.
10. E. Karbab, M. Debbabi, A. Derhab, D. Mouheb, Android Malware Detection using Deep Learning on API Method Sequences, arXiv.org e-Print archive, 2017, https://arxiv.org/pdf/1712.08996.pdf
11. R. A. Nix. Applying Deep Learning Techniques to the Analysis of Android APKs, digitalcommons.lsu.edu: Louisiana State University Digital Commons, 2016, https://digitalcommons.lsu.edu/cgi/viewcontent.cgi?article=5443&context=gradschool_theses.
12. N. Peiravian, X. Zhu, Machine Learning for Android Malware Detection Using Permission and API Calls, ICTAI '13 Proceedings of the 2013 IEEE 25th International Conference on Tools with Artificial Intelligence, pp. 300-305.
13. Ignatev G. Y., Pavlenko E. Y., Using machine learning for detection of malicious Android applications, Information security of Russian regions, Conference materials, 2017, pp. 101-103.
14. K. Au, Y. Zhou, Z. Huang, D. Lie, PScout: Analyzing the Android Permission Specification, In the Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS 2012) Association for Computing Machinery (ACM), pp. 217-228.
15. M. Backes, S. Bugiel, E. Derr, P. McDaniel, D. Octeau, S. Weisgerber. On Demystifying the Android Application Framework: Re-Visiting Android Permission Specification Analysis, 2016, https://pdfs.semanticscholar.org/361d/0121f49d526602ddd9bf369a4bc402e5e0fb.pdf.
16. MurmurHash3, https://github.com/aappleby/smhasher/wiki/MurmurHash3.
17. Android Drawer: Android Apps .apk Repository, https://www.androiddrawer.com/.
18. Playdrone-apk-e8, https://archive.org/details/playdrone-apk-e8.
19. E. Mariconti, L. Onwuzurike, P. Andriotis, E. De Cristofaro, G. Ross, G. Stringhini, MaMaDroid: Detecting Android Malware by Building Markov Chains of Behavioral Models, arXiv.org e-Print archive, 2017, https://arxiv.org/pdf/1612.04433.pdf.
20. E. Karbab, M. Debbabi, A. Derhab, D. Mouheb, Android Malware Detection using Deep Learning on API Method Sequences, arXiv.org e-Print archive, 2017, https://arxiv.org/pdf/1712.08996.pdf.
21. Android Malware Dataset, http://amd.arguslab.org/.
22. Androguard, https://github.com/androguard/androguard.
23. A. Krizhevsky. CIFAR-10 and CIFAR-100 datasets, https://www.cs.toronto.edu/~kriz/cifar.html.
24. Keras: The Python Deep Learning library, https://keras.io/.
25. Paperspace, https://www.paperspace.com/.
26. Imbalanced-learn, https://github.com/scikit-learn-contrib/imbalanced-learn/.