Evaluating Opcodes for Detection of Obfuscated Android Malware

Saneeha Khalid
Computer Science Department
Bahria University
Islamabad, Pakistan
01-284172-002@student.bahria.edu.pk

Faisal Bashir Hussain
Computer Science Department
Bahria University
Islamabad, Pakistan
fbashir.buic@bahria.edu.pk

***Abstract*—Obfuscation refers to changing the structure of code in a way that original semantics can be hidden. These techniques are often used by application developers for code hardening but it has been found that obfuscation techniques are widely used by malware developers in order to hide the work flow and semantics of malicious code. Class Encryption, Code Re-Ordering, Junk Code insertion and Control Flow modifications are Code Obfuscation techniques. In these techniques, code of the application is changed. These techniques change the signature of the application and also affect the systems that use sequence of instructions in order to detect maliciousness of an application. In this paper an 'Opcode sequence' based detection system is designed and tested against obfuscated samples. It has been found that the system works efficiently for the detection of non obfuscated samples but the performance is effected significantly against obfuscated samples. The study tests different code obfuscation schemes and reports the effect of each on sequential opcode based analytic system.**

***Index Terms*—obfuscation, opcodes, malware, LSTMs**

I. Introduction

Android is the most used mobile operating system at present. It provides its users with a large number of useful and entertaining applications. These applications can be downloaded from official Google play store. Although the applications present on the play-store are very beneficial for users; but on the other hand malware writers also create applications with malicious functions. The rate of malware penetration among android applications has increased in recent past. 10.5 million android malware were found in 2019 and 0.48 million new malware were found in 2020 [1].

These malwares also deploy obfuscation schemes which make their detection more difficult. Obfuscation is a method for changing the structure of code in a way that the original structure is either hidden or changed[12]. Malware writers use obfuscation for generating variants of malwares that have been registered by anti-malware products[2]. This makes them difficult to be detected by commercial anti-malware products. It has been reported by many studies that performance of anti-malware products decrease by a significant percentage against obfuscated samples. [16] [7] [5].

Code Obfuscation techniques are also very effective in deceiving the detection schemes that work on the pattern of code to identify malicious applications. Over the time many schemes have been developed that focus on the analysis of code for generation of features. Many studies [1], [4] and [9] have used code based features and have reported high accuracy for detection of android malware. However it has been reported by [4] that code obfuscation techniques can greatly influence the performance accuracy of systems that work on code based features.

Class encryption, Junk code insertion, Control Flow modifications, and Code Re-ordering are popular code obfuscation techniques[14]. In this paper the effect of these obfuscation techniques on the performance of systems that use opcode based features for designing the classification engine has been critically analyzed. In order to perform this analysis; a classifier that works on opcode based features is designed. It has been observed that the system works well with non obfuscated samples but the efficiency of the system degrades when obfuscated samples are tested. The paper tests four code based obfuscation schemes and reports the efficiency of opcode based system against each obfuscation scheme. Following are the major contributions of the paper:

- An 'opcode sequence' based malware detection engine has been designed. One-gram and two-gram opcode sequences are extracted and LSTM network is trained. This study has analyzed the efficiency of opcodes as features for 2 class (Malicious and Benign) and 4 class (Adware, Ransom, Banking Trojan and Benign) problems.
- Data set for Obfuscated samples is generated. DashO[2] and Obfuscapk[3] is used for applying code obfuscations to malicious samples. Control Flow modifications , Code Reordering and Junk Code Insertions are applied. PraGuard data set is used for Class Encryption based samples.
- The designed system is tested against each obfuscation scheme and results are reported.

The remainder of the paper is organized as follows. Section 2 describes the related work. Section 3 describes the rationale

[1]https://www.statista.com/statistics/680705/global-android-malware-volume/

[2]https://www.preemptive.com/products/dasho
[3]https://github.com/ClaudiuGeorgiu/Obfuscapk

for the study. Section 4 describes the proposed evaluation framework. Section 5 describes the results obtained on obfuscated and non-obfuscated samples and finally conclusion from the study is presented in section 6.

II. Related Work

This section covers the studies that have used opcode sequences for classification of malicious applications. Amin et al.[1] developed an end to end opcode based system. Dex byte codes are extracted and used to predict the maliciousness of an application. Bi directional LSTM networks are deployed and opcodes are represented using one hot encoding. They have evaluated their results on Drebin , AMD and virus share data sets. However results have not been tested on Obfuscated samples. Pektas et al. [10] extracted instruction call sequences from call graphs and applied deep learning models like LSTMS and CNNs for malware classification. An accuracy of 91.42 is reported but the data set does not contain obfuscated malicious applications.

Chen et al. [4] performed a detailed analysis on the selection of opcodes as features. Only important opcodes are selected and then sequence is formulated. 3-gram opcode sequences are used. Classification is performed using using Random Forest and SVM. Authors have stated it clearly that their system is vulnerable against metamorphic malware samples. Authors in [6] and [15] extracted opcodes from classes.dex file. Opcode patterns are used as images and CNN based networks are used for classification. Authors in [9] worked on raw opcode sequences extracted from dex files. 1-gram , 2-gram and 3-gram sequences have been extracted and tested. CNNs are used for classification.

III. Rationale for Current Research

Some existing studies considered the problem of analyzing the effect of obfuscation on the detection schemes. In this section, the proposed system is compared with existing schemes and the merits of our scheme are discussed. Hammad et al. [7] have analyzed the effect of obfuscation on the performance of anti viruses. It is reported that performance of many anti viruses degrades when obfuscated samples are tested. However they have not analyzed any particular technique used by a detection engine which is specifically affected by a certain obfuscation technique.

Bacci et al. [3] have studied the impact of code obfuscation on static and dynamic machine learning based techniques. They have considered opcodes frequency and system calls as features. It is reported that obfuscation effects the static analysis based methods more adversely than dynamic analysis based methods. This study is the close to our work , however our study has specifically focused on the performance degradation of sequential deep classifiers.

To the best of our knowledge, this is the first work which has investigated the effect of Code obfuscation schemes on the performance of deep sequential classifiers given static feature set. We have explained the effect of each code obfuscation scheme on the sequential classification systems and have also reported the performance degradation metrics against each obfuscation scheme separately.

IV. Proposed Evaluation Framework

The purpose of our evaluation framework is to measure the effectiveness of opcode based analytic system on non-obfuscated and obfuscated samples. For this purpose an ’opcode sequence’ based analytic system is designed. The system extracts opcode sequences from an application and a classification engine is trained. The samples for training include non-obfuscated malware samples. Later code obfuscation schemes that effect opcode sequences of the android applications are applied on the malware samples. The resultant obfuscated data set is then tested on the designed system and accuracy is measured. The complete experimental setup is depicted in Figure 1.

This section is arranged as follows. First subsection represents the design of the opcode based detection system. Second subsection discusses code obfuscation schemes in detail. Data set preparation and testing on obfuscated data set is also discussed in second sub section.

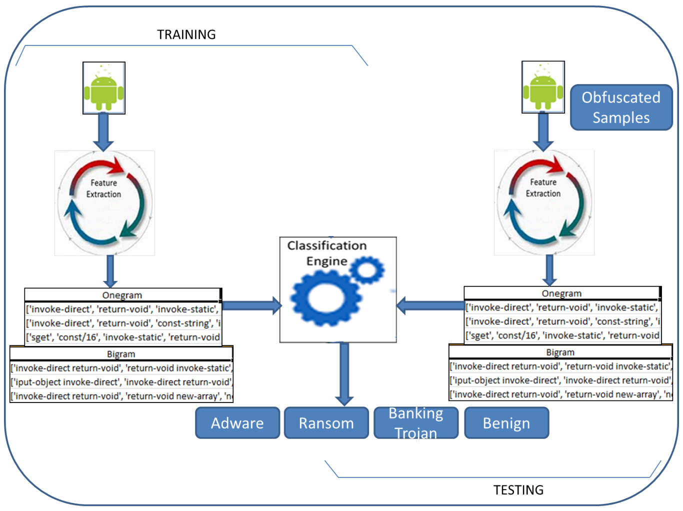
TRAINING
Obfuscated Samples
Feature Extraction
Feature Extraction
Classification Engine
Onegram
['invoke-direct', 'return-void', 'invoke-static',
['invoke-direct', 'return-void', 'const-string', 'i
['sget', 'const/16', 'invoke-static', 'return-void
Bigram
['invoke-direct return-void', 'return-void invoke-static',
['iput-object invoke-direct', 'invoke-direct return-void',
['invoke-direct return-void', 'return-void new-array', 'n
Onegram
['invoke-direct', 'return-void', 'invoke-static',
['invoke-direct', 'return-void', 'const-string', 'i
['sget', 'const/16', 'invoke-static', 'return-void
Bigram
['invoke-direct return-void', 'return-void invoke-static',
['iput-object invoke-direct', 'invoke-direct return-void',
['invoke-direct return-void', 'return-void new-array', 'n
Adware
Ransom
Banking Trojan
Benign
TESTING

Fig. 1. Evaluation Framework for obfuscated and non-obfuscated applications

A. *Design of Opcode based analytic system*

Opcodes represent the instructions used in an application. A sequence of opcodes is a better representative of an application’s behavior. Many studies [1] [10] and [4] have created detection systems based on opcode sequences. In this section the design details of the proposed opcode based detection system are discussed. The discussion includes the selected sources of data for non obfuscated malicious applications, the process of feature extraction and the application of classifier.

1) Data Set: The malicious and benign applications used for this study are obtained from CICMalDroid2020[4] and Androzoo[5] data sets. The data is divided into groups. The

[4]https://www.unb.ca/cic/datasets/maldroid-2020.html
[5]https://androzoo.uni.lu/

TABLE I
RELATED WORK

Paper Name	Features	Classifier	Obfuscation Tested	Accuracy
Amin et al.[1]	dex bytecode	bi-dir LSTM	No	99.6
Pektas et al. [10]	instruction calls	LSTM, CNN	No	91.6
Chen et al. [4]	3-gram sequences	RandomForest	No	95.3
Ren et al. [13]	dex bytecode	DexCRNN	N0	93.4
McLaughlin et al. [9]	2-gram,3-gram raw bytecode	CNN	No	95

first group contains samples of 4 categories; Adware, Ransom, Banking Trojans and Benign. In the second group data samples are labeled as malicious and benign only. The purpose of creating two groups is to verify the effectiveness of Opcode sequences for classification of 4 class and 2 class problems.

2) Feature Extraction: The features used in this experiment are opcode sequences. For extraction of opcodes; the apk file is first disassembled into smali files. After disassembling; the smali files are parsed for selecting the opcodes. A standard list of opcodes is used as a reference. Each extracted line from the smali file is compared with the standard opcode list and standard opcodes used in the apk are extracted.

The process is repeated for each smali file in the apk. It must be noted that opcodes are extracted in two formats; one gram and two grams. Accordingly the experiments are performed against both type of opcode sequences separately. Also note that these csvs are maintained and labeled separately for four class and two class problems.

3) Application of Classifier: LSTMs are an extension of RNN networks and have shown promising results in prediction of sequence data. LSTMs are widely used in other domains like speech recognition very successfully. Qui et al. [11] analyzed many studies on android malware detection and found that LSTMs usage with sequential features like opcodes or system calls traces is efficient for malware classification and categorization. Therefore LSTMs have been chosen as the classifier for this study. The input to the LSTM model is the sequence of one gram and two gram opcodes for an apk. The length of each sequence varies as different apks have different number of opcodes.

Keras sequential model is used for the design of classifier. The sequential model consists of 100 LSTM layers, one Dropout layer (for preventing over fitting), one Flatten layer and finally Dense layers. Rectified Linear Unit is used as the activation function in the Dense layers. The model is compiled with adam as optimizer and categorical cross entropy as loss function. The design of classification engine is presented in Figure 2. The compiled model is then trained on the input data. In the training phase number of epochs are set to 100 and a batch size of 128 is used.

In order to make our work helpful for other researchers; we have made the code of our system public [6]. The performance of the system in terms of training and testing accuracy is discussed in section 4.

[6]https://github.com/saneeha-amir/Obfuscated-Android-Malware-Detector

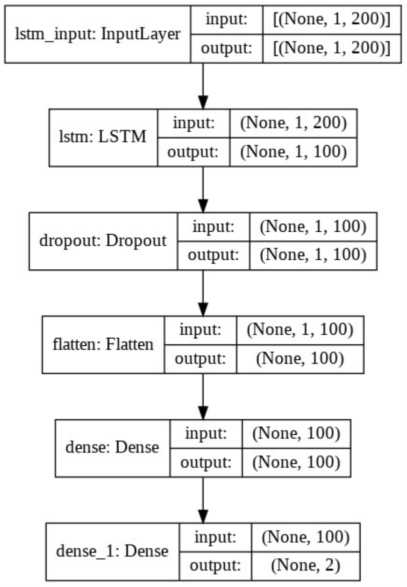

Fig. 2. Design of Classification Engine

B. Code Obfuscation

Code obfuscation refers to changing the structure of code in order to hide the semantics. For this purpose specific classes and methods need to accessed and changes are to be made. In order to perform code modifications, the application first needs to be disassembled. Apktool is a famous tool for disassembling of apk file. After disassembling of apk, the class files in the android application are converted into smali files; which contain assembly language instructions. The code changes are then made in the smali file. After updating the smali files the application is repacked and signed.

1) Obfuscation schemes effecting the regular opcode structure: Many obfuscation schemes effect the code structure of an application. This study has focused on the obfuscation schemes that particularly effect the opcode sequences of the apk file. Different obfuscation schemes are analyzed and the schemes that particularly effect the opcode sequencing are selected. The selected schemes include:

- Class encryption
- Junk code insertion
- Control flow Obfuscation
- Code re-ordering

a) Class Encryption: Class encryption is an obfuscation technique in which the class files of the android application are encrypted. In this way the code becomes unavailable for static analysis. It is the most powerful obfuscation technique[8]. This technique changes the hash of file and also effects all static code based analytic methods. Class encryption strongly effects the opcode based detection scheme as the extracted opcodes are meaningless due to encryption. DexGuard supports class encryption. It is a paid tool and is not available for educational purposes. However PraGurad data set contains obfuscated samples for class encryption.

b) Junk code-Insertion - Non-functional methods: Junk code insertion is one the code obfuscation techniques. In this technique some non-functional code is inserted into the application. Insertion of non-functional methods is one of the techniques for junk code insertion. Non-functional methods may perform some trivial functionality like printing a string. Addition of these methods do not effect the overall working but effects the method table in the Dalvik byte code [17]. As a result the signature of application is also changed. ADAM is the tool that supports this technique for junk code insertion. The tool parses the smali files and inserts non functional methods before the constructor.

c) Junk code-Insertion - 'No Operation'(Nop) Instructions: Nop is a valid opcode that does noting. It is a trivial junk code insertion technique. By inserting NOP instructions the hash of the file is changed. The sequence of original opcodes is also changed by adding NOP opcodes in different methods. Obfuscapk[2] is the tool that supports the insertion of NOP in different methods of applications. This tool parses all the smali files of the application and inserts a random number (1 - 5) of nop opcodes after a valid opcode in every method.

d) Junk code-Insertion - Over Loaded Methods: Method overloading is a useful feature of Java programming language. In order to add junk code inside application; overloaded method insertion can also be used. The insertion of overloaded methods change the hash of the application and also effects the opcode based static analysis systems. Obfuscapk [2] supports the insertion of overloaded methods. In order to insert an overloaded method; the classes in each smali file are read. The methods in the classes are analyzed and overloaded versions are generated. The overloaded methods have different number of arguments and void return type. The body of these methods is then filled with some arithmetic instructions.

e) Junk code-Insertion of Arithmetic Branch: Arithmetic branch is a path based on the result of some arithmetic operation. They can be used as Junk code if the arithmetic operation is never true. Insertion of arithmetic branches changes the hash of code and sequence of opcodes. Obfuscapk [2] supports the insertion of arithmetic branches. The tool parses the smali files and locates the methods which are not abstract. Inside these methods an addition and remainder operation is inserted. 'if' condition is applied and a goto instruction is inserted in the else part which is never taken.

f) Control flow Obfuscation: In control flow based obfuscation techniques; the control flow of application is changed by inserting iterative structures, goto statements or code branching instructions. All these techniques change the original opcode sequence of the application. In order to apply the control flow based obfuscation techniques; application is broken to smali and then changes are applied and later app is rebuild and resigned. Obfuscapk , DashO , Allotari are some of the tools that support control flow obfuscation.

g) Code re-ordering: In code re ordering , the order of the instructions and methods is changed. As the code is re-ordered; the hash of the file changes and the sequence of opcodes also changes. This technique is also applied on the disassembled smali files. Obfuscapk supports code reordering obfuscation.

2) Data set for Obfuscated Samples: For the generation of obfuscated samples DashO and Obfuscapk have been used. DashO is a commercial tool and only trial version is available. Obfuscapk is an open source tool and is available on gitHub. Code reordering, Junk Code Insertion and Control Flow modifications are applied on the data set using Obfuscapk and DashO. Samples for class encryption have been obtained from PraGuard[7] data set.

3) Testing on Classifier: After the generation of obfuscated samples; two-gram opcode sequences are extracted against each sample. A csv file is maintained against each obfuscation category. Each category of samples is tested against the trained classification engine and it has been noted that different obfuscation schemes have different effect on the performance of the system. The results of testing are presented in section 4.

V. Results and Discussion

In this section the accuracy of the designed system on obfuscated and non-obfuscated applications is shown. It has been observed that opcode sequences are meaningful features as the system predicts the 4 class category samples with an accuracy of 92.5 percent and 2 class category is predicted with an accuracy of 97.2 percent. Hence it can be concluded that opcode sequences when used with a powerful sequence based classifier like LSTM forms an efficient malware detection system. The training and validation accuracy of the system is presented in Figures 3, 4, 5 and 6.

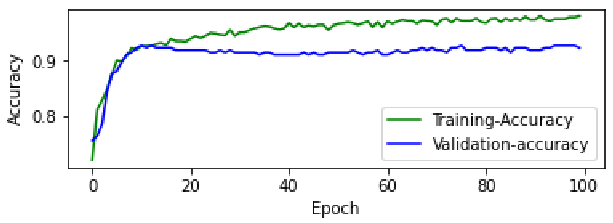

Fig. 3. One-Gram 2 class Model Accuracy

[7]http://pralab.diee.unica.it/en/AndroidPRAGuardDataset

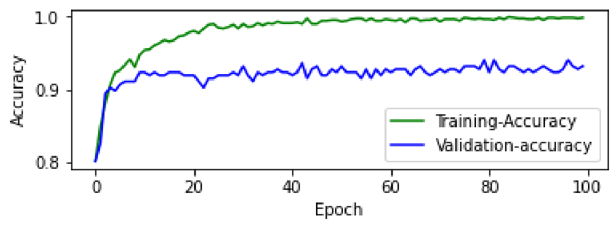

Fig. 4. Two-Gram 2 class Model Accuracy

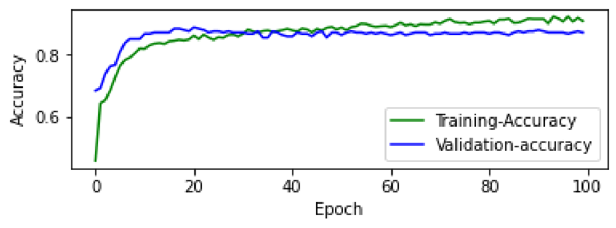

Fig. 5. One-Gram 4 class Model Accuracy

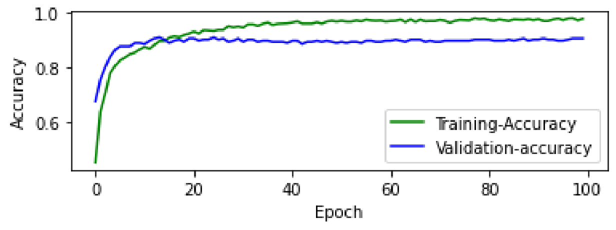

Fig. 6. Two-Gram 4 class Model Accuracy

The performance of system on non-obfuscated test data is shown in table II.

TABLE II
CLASSIFICATION ACCURACY FOR 1-GRAM AND 2-GRAM OPCODES

Type of Data	Features Type	Precision	Recall	F-Score
Non-Obfuscated 4 class	one-gram	0.88	0.87	0.874
Non-Obfuscated 4 class	two-gram	0.90	0.91	0.904
Non-Obfuscated 2 class	one-gram	0.93	0.935	0.932
Non-Obfuscated 2 class	two-gram	0.94	0.95	0.944

The trained system is then tested on obfuscated samples. As mentioned in previous section that a data set containing obfuscated samples was created for testing the designed system. The data set contains samples for Junk Code insertion techniques, Code re-ordering, Code encryption and class reordering obfuscation techniques. After the generation of data set; feature extraction is run on these samples. 2 gram opcode sequences are extracted from samples against each obfuscation scheme. The resultant feature set is then tested on the trained LSTM network. Following evaluation metrics are used for reporting the performance of designed system on obfuscated samples:

- Accuracy : The percentage of correctly identified both positive and negative samples

TABLE III
EFFECT OF OBFUSCATION

Obfuscation Technique	FPR (False Positive Rate)	FNR (False Negative Rate)
Class Encryption	0.83	0.80
JNK(Nop Opcode)	0.16	0.05
JNK (De-Functional methods)	0.30	0.20
JNK (Over Loaded methods)	0.28	0.25
JNK (Arithmetic Branches)	0.16	0.15
Control Flow	0.35	0.27
Code Re-ordering	0.20	0.20

- FPR (False Positive Rate): Rate of incorrectly predicting positive class
- FNR (False Negative Rate): Rate of incorrectly predicting negative class

The accuracy obtained with obfuscated samples is shown in Figure 7. The values for False Positive Rates (FPR) and False Negative Rates (FNR) are listed in Table III. It has been observed that the performance of the system is effected when obfuscated applications are tested. Different obfuscation schemes effect the working of the system in different ways.

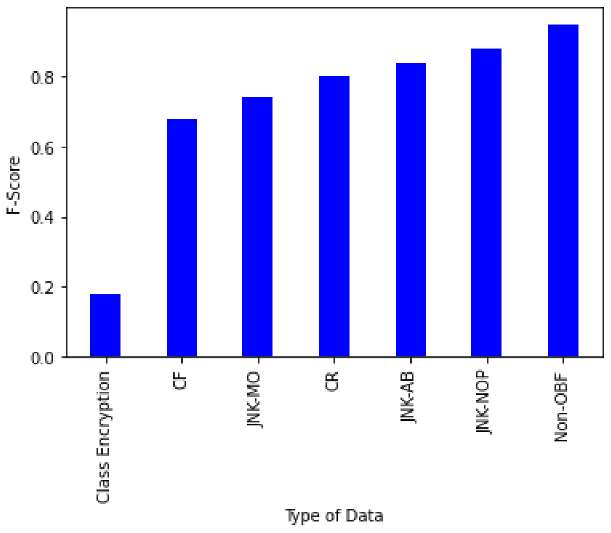

Fig. 7. Accuracy Variations with Obfuscated Samples

The performance of the classification engine is most effected by class encryption. As complete class is encrypted; therefore the semantic of opcodes is no longer preserved. So the efficiency of the system drops by a significant percentage. After class encryption; Control Flow obfuscation effects the working of system most. Here the opcode sequence is effected by adding goto statements, for loops and conditional statements. By adding these structures the opcode sequence is changed and therefore the efficiency of the system drops by a significant percentage.

Detection of samples with Code reordering is also effected by some percentage. Here the data and methods inside the classes are reordered and hence the sequence is altered. Junk code schemes of Inserting overloaded and de-functional methods also effect the schemed by a significant percentage.

Junk code schemes of NOP insertion and Arithmetic branching have lesser impact on the efficiency of the system.

It must be noted that 2-gram opcode sequences are used for testing of obfuscated samples. From these results it can be concluded that 'opcode sequence' based detection schemes are very efficient for the prediction of malicious applications but when obfuscated samples are supplied; their performance gets effected. From the results of this study it can be concluded that static opcode based techniques become less efficient against code obfuscations.

However we suggest that run time extraction of opcodes can help in overcoming this problem as many obfuscations schemes become ineffective when code is analyzed at run time. For example in case of class encryption; decrypted code can be extracted and analyzed dynamically. Similarly the control flow modifications effect the static analysis techniques only as the change of flow is not actually executed at run time. Same is the case with arithmetic branch insertions and over loaded method insertions as these branches and methods are never executed. Run time extraction and analysis of code can help the analytic engine to focus on the actual code of the application and can fade away the effects generated by obfuscation.

VI. Conclusion

In this study an 'opcode sequence' based analytic engine is designed. The system is trained and tested on non obfuscated samples and the results are very promising. But it has been observed that the efficiency of the system drops significantly when samples with different obfuscations are supplied. From this study it can be concluded that opcodes are an efficient feature set specially when used in form of sequence. In order to formulate obfuscation resilient system, it is proposed that code based features like opcodes should be collected dynamically at run time so that the effect of obfuscation is minimized.

References

[1] Muhammad Amin, Tamleek Ali Tanveer, Mohammad Tehseen, Murad Khan, Fakhri Alam Khan, and Sajid Anwar. Static malware detection and attribution in android byte-code through an end-to-end deep system. *Future Generation Computer Systems*, 102:112–126, 2020.

[2] Simone Aonzo, Gabriel Claudiu Georgiu, Luca Verderame, and Alessio Merlo. Obfuscapk: An open-source black-box obfuscation tool for android apps. *SoftwareX*, 11:100403, 2020.

[3] Alessandro Bacci, Alberto Bartoli, Fabio Martinelli, Eric Medvet, Francesco Mercaldo, and Corrado Aaron Visaggio. Impact of code obfuscation on android malware detection based on static and dynamic analysis. In *ICISSP*, pages 379–385, 2018.

[4] Tieming Chen, Qingyu Mao, Yimin Yang, Mingqi Lv, and Jianming Zhu. Tinydroid: a lightweight and efficient model for android malware detection and classification. *Mobile information systems*, 2018, 2018.

[5] Melissa Chua and Vivek Balachandran. Effectiveness of android obfuscation on evading anti-malware. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, pages 143–145, 2018.

[6] Abdulbasit Darem, Jemal Abawajy, Aaisha Makkar, Asma Alhashmi, and Sultan Alanazi. Visualization and deep-learning-based malware variant detection using opcode-level features. *Future Generation Computer Systems*, 125:314–323, 2021.

[7] Mahmoud Hammad, Joshua Garcia, and Sam Malek. A large-scale empirical study on the effects of code obfuscations on android apps and anti-malware products. In *Proceedings of the 40th International Conference on Software Engineering*, pages 421–431, 2018.

[8] Davide Maiorca, Davide Ariu, Igino Corona, Marco Aresu, and Giorgio Giacinto. Stealth attacks: An extended insight into the obfuscation effects on android malware. *Computers & Security*, 51:16–31, 2015.

[9] Niall McLaughlin, Jesus Martinez del Rincon, Boo-Joong Kang, Suleiman Yerima, Paul Miller, Sakir Sezer, Yeganeh Safaei, Erik Trickel, Ziming Zhao, Adam Doupé, et al. Deep android malware detection. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, pages 301–308, 2017.

[10] Abdurrahman Pektaş and Tankut Acarman. Learning to detect android malware via opcode sequences. *Neurocomputing*, 396:599–608, 2020.

[11] Junyang Qiu, Jun Zhang, Wei Luo, Lei Pan, Surya Nepal, and Yang Xiang. A survey of android malware detection with deep neural models. *ACM Computing Surveys (CSUR)*, 53(6):1–36, 2020.

[12] Vaibhav Rastogi, Yan Chen, and Xuxian Jiang. Catch me if you can: Evaluating android anti-malware against transformation attacks. *IEEE Transactions on Information Forensics and Security*, 9(1):99–108, 2013.

[13] Zhongru Ren, Haomin Wu, Qian Ning, Iftikhar Hussain, and Bingcai Chen. End-to-end malware detection for android iot devices using deep learning. *Ad Hoc Networks*, 101:102098, 2020.

[14] Kimberly Tam, Ali Feizollah, Nor Badrul Anuar, Rosli Salleh, and Lorenzo Cavallaro. The evolution of android malware and android analysis techniques. *ACM Computing Surveys (CSUR)*, 49(4):1–41, 2017.

[15] Junwei Tang, Ruixuan Li, Yu Jiang, Xiwu Gu, and Yuhua Li. Android malware obfuscation variants detection method based on multi-granularity opcode features. *Future Generation Computer Systems*, 129:141–151, 2022.

[16] Yinxing Xue, Guozhu Meng, Yang Liu, Tian Huat Tan, Hongxu Chen, Jun Sun, and Jie Zhang. Auditing anti-malware tools by evolving android malware and dynamic loading technique. *IEEE Transactions on Information Forensics and Security*, 12(7):1529–1544, 2017.

[17] Min Zheng, Patrick PC Lee, and John CS Lui. Adam: an automatic and extensible platform to stress test android anti-virus systems. In *International conference on detection of intrusions and malware, and vulnerability assessment*, pages 82–101. Springer, 2012.