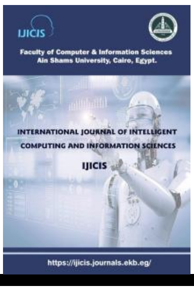
IJICIS, Vol.24, No.1, 1-11
DOI:10.21608/ijicis.2024.250295.1308

International Journal of Intelligent Computing and Information Sciences

https://ijicis.journals.ekb.eg/


IJICIS
Faculty of Computer & Information Sciences Ain Shams University, Cairo, Egypt.
INTERNATIONAL JOURNAL OF INTELLIGENT COMPUTING AND INFORMATION SCIENCES
IJICIS
https://ijicis.journals.ekb.eg/

ARTIFICIAL INTELLIGENCE BASED ALGORITHM FOR DETECTING ANDROID OBFUSCATED APPLICATIONS

Hend Faisal*
Faculty of Computer and Information Sciences, Ain Shams University,Egyptian Computer Emergency and Readiness Team (EG-CERT), National Telecom, Regulatory Authority (NTRA), Egypt
hend.faisal@cis.asu.edu.eg

Hanan Hindy
Faculty of Computer and Information Sciences, Ain Shams University,
Cairo, Egypt
hanan.hindy@cis.asu.edu.eg

Samir Gaber
Faculty of Engineering in Helwan, Helwan University
Egyptian Computer Emergency and Readiness Team (EG-CERT), National Telecom,
Regulatory Authority (NTRA), Egypt
Samir_abdelgawad@eng-helwan.edu.eg

Abdel-Badeeh M. Salem
Faculty of Computer and Information Sciences, Ain Shams University,
Cairo Egypt
absalem@cis.asu.edu.eg

Received 2023-11-22; Revised 2023-11-22; Accepted 2024-03-05

Abstract: *As technology continues to advance, so does the landscape of Android; based on its open-source nature which renders it vulnerable to various risks. Therefore, the developers need to deploy and employ obfuscation techniques in their newly developed android applications. In this paper, we present an investigation into Android obfuscation detection. Our work encompasses a comprehensive examination of Android obfuscation techniques and an exploration of their intersection with machine learning. We conducted extensive experiments involving various machine learning models to detect obfuscation. The results show that Random Forest is the one with the most promising results with accuracy* ***99.5%*** *in detecting Android Obfuscation. The used dataset comprises both malicious and benign samples, where different obfuscation techniques have been applied.*

Keywords: *Android Obfuscation, Machine Learning, Artificial Intelligence, Information Security*

1. Introduction

Android holds significant importance in the market as it is the most used platform according to Worldwide Market Share statistics [1]. As a result of the nature of Android operating system which is based on open-source development, Android application developers suffer from how easy their applications could be reverse-engineered, and thus, their code being accessible and readable. This

***Corresponding Author**: Hend Faisal

Faculty of Computer and Information Sciences, Ain Shams University, Egyptian Computer Emergency and Readiness Team (EG-CERT), National Telecom, Regulatory Authority (NTRA), Egypt

Email address: hend.faisal@cis.asu.edu.eg

accessibility makes the applications vulnerable to code theft from individuals to build their own applications based on others' work. Also, this allows malicious actors to copy and rewrite an application customized for their own malicious purposes. As a result, the need for a technique to safeguard sensitive data became critical, and code obfuscation is one way of doing it. Code Obfuscation [2] helps in doing this by offering a layer of protection against reverse engineering and code analysis.

Code obfuscation is a technique used in making the source code of an application more difficult to be analyzed and reduce the readability of the code. Besides its benefits, obfuscation is a two-edged sword. For legitimate software, it provides an additional layer of security to help in preventing unauthorized access to their critical components. On the other hand, for malicious software, it poses challenges for security professionals in detecting malware by using automated malware detection tools as it hides its malicious activities [20]. The majority of the current research proposed detection approaches that focus on detecting each obfuscation technique separately.

In this work, we proposed a framework for detecting android obfuscation with more than one technique. A comparative analysis is conducted to evaluate and contrast the performance of various machine learning algorithms in order to ascertain which one offers the most effective means of detecting the different obfuscation techniques. Furthermore, the feature importance is evaluated for the obfuscation detection problem. The proposed model efficiency is assessed with two different datasets described in detail in the Dataset section.

We organized our paper as follows: The introduction is presented in section 1, background and explaining the nature of Android Application Package (APK) and how source code is compiled to an APK, how obfuscation is done in android and various obfuscation tools in the market in section 2. Section 3 reviews related work in the field. Section 4 outlines our research methodology, including data preprocessing, dataset, evaluation metrics, and results. Finally, the paper concludes in Section 5, summarizing the findings and pointing to future directions in the realm of Android obfuscation detection.

2. Background

Understanding the internal components of APK [3] is crucial for understanding how reverse engineering is easy for Android applications if they are not obfuscated. APK files are considered a zip file that encapsulates different components as 1) Android Manifest, which contains essential information about the application including its version, permissions, activities and more. 2) META-INF, which includes application signature and integrity-related information. 3) classes.dex, a file that contains the compiled Dalvik bytecode of the app. 4) Resource folder, which contains non-compiled resources that the application needs. 5) Assets, which include application assets if needed like storing data, configuration files, HTML files, or other resources that the application might need. 6) Lib directory, that holds compiled native libraries for various CPU architectures. The compilation process from source code to the Dalvik Virtual Machine involves several steps in the Android application development workflow.

As shown in Figure 1, the **Source code** is writing the code of the Android application in one of its supported languages (i.e., Java or Kotlin).
Bytecode is considered an intermediary component between high-level source code and runtime execution where the source code is compiled into java bytecode.

Dalvik Bytecode is an optimized representation of Android code and stored in .dex format.
Dalvik Virtual Machine is responsible for executing Dalvik bytecode on android devices.

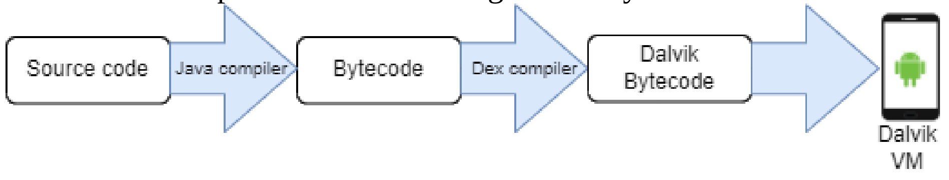


Figure. 1: Compilation steps of Android Application

As a reverse engineer, it is straightforward to decompile an Android application and retrieve all of those components and easily decompile Dalvik bytecode to easily human-readable source code by various tools such as APKtool [4], Dex2jar [5] and more. Thus, understanding all the functionalities of the app. Here came the importance of using obfuscation.

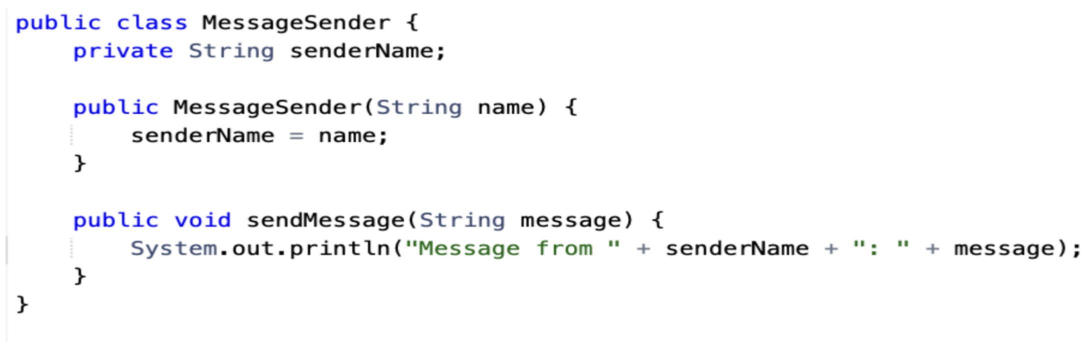
2.1. Obfuscation in Android

Obfuscation helps to prevent reverse-engineering attempts by making it harder for attackers to understand the source code of the app, control flow and its resource structure. There exist multiple obfuscation techniques.

2.1.1 Obfuscation Techniques

In this section, the different Android Obfuscation techniques are outlined:

- ***Identifier Renaming:*** involves changing the names of classes, fields and methods of source code to be less descriptive so make it harder for attackers to comprehend the application's logic.

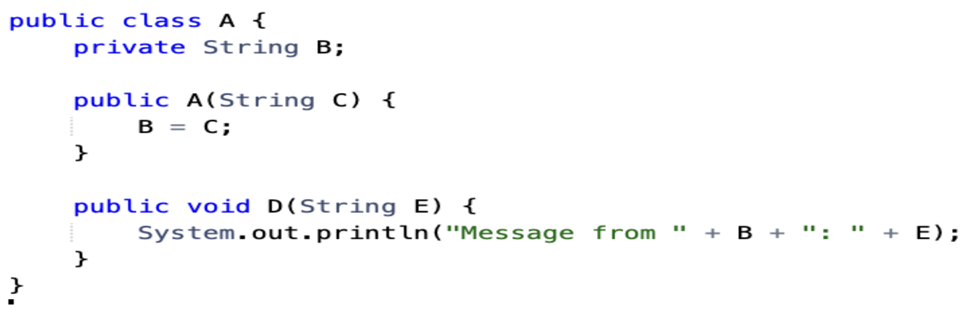


```
public class MessageSender {
    private String senderName;

    public MessageSender(String name) {
        senderName = name;
    }

    public void sendMessage(String message) {
        System.out.println("Message from " + senderName + ": " + message);
    }
}
```

Figure. 2.a: Code snippet before Identifier Renaming



```
public class A {
    private String B;

    public A(String C) {
        B = C;
    }

    public void D(String E) {
        System.out.println("Message from " + B + ": " + E);
    }
}
```

Figure. 2.b: Code snippet after Identifier Renaming

- ***String/ Class Encryption:*** In this technique, the developers encrypt strings and/or classes instead of saving strings as plain text and decrypt them at runtime. For example,

  Text before encryption: "Hello, World!"
  Text after encryption: "5$K7#p2!ZaP&L9*"

- ***Control Flow Obfuscation:*** This technique involves modifying the flow of the code thus making it more complex and harder to follow. Examples of code flow obfuscation include [19]:
  - ***Junk code insertion*** which involves injecting redundant and irrelevant code into the original codebase to obfuscate the logic of the application.
  - ***Reflection*** involves accessing dangerous APIs of the Android Framework to employ dynamic code execution to invoke APIs at running time.
  - ***Call indirection*** which hides the direct calls to main functions to obscure the actual flow of the application.
  - ***GoTo/ Nop instructions insertion*** that disrupts the natural flow of the program either by adding jumps or not instructions.
  - ***Reorder code blocks*** involves altering the sequence of code sections within the application
- ***Resource Encryption:*** *which* encrypts critical resources and assets to prevent unauthorized access.

2.1.2 Obfuscation Tools

There exists a number of obfuscation tools that have been developed lately which made obfuscation easy to be used. Table 1 lists the top five used obfuscation tools and their proposed features.

Table 1: Popular Android Obfuscation Tools

Obfuscator	Features	Opensource
ProGuard [6]	Identifier Renaming APK Shrink Bytecode Optimization	✓
DexGuard [7]	Identifier Renaming APK Shrink Bytecode Optimization Control Flow Obfuscation Class/ String/ Resource Encryption	✗
Allatori [8]	Identifier Renaming Control Flow Obfuscation String Encryption	✗
DashO [9]	Identifier Renaming Control Flow Obfuscation String/ Resource Encryption	✗
ObfuscAPK [10]	Identifier Renaming Control Flow Obfuscation String/ Class/ Resource Encryption	✓

3. Related Work

In the literature, there are several research papers working on the Android obfuscation problem. Jiang *et.al.* [11] proposed a function-level obfuscation detection method based on a hybrid Neural Network model; Graph Convolutional Network (GCN) and Long Short-Term Memory (LSTM) (GCN-LSTM). Their proposed architecture comprised both x86 architecture and Android applications. The authors used the same approach for both architectures, but each one separately due to the their difference. They extracted 15 features using IDAPython plugin provided by IDAPro and built multiple Machine Learning model including Adaboost, GaussianNaiveBayes, GradientBoosting, etc. Their findings showed that the one with the best accuracy was GCN-LSTM with accuracy **98.94%**.

Park *et.al.* [12] targeted the problem from another point of view. The authors found out that most of the researchers identify obfuscation at application-level and they claim that this approach is limited and inaccurate. Therefore, they started to employ the idea of class level obfuscation as they decompile the APK, then extracted the classes from classes.dex and vectorized the decompiled code to output vector which they fed into four different machine learning classifiers and detect four obfuscation techniques. Extra Trees algorithm was the best upon others used algorithms with average accuracy ***75.9%***

Mirzaei *et.al.* [13] presented Androdet, an obfuscation detector implemented to detect three types of obfuscation renaming identifier, string encryption, and control flow obfuscation. They extracted 30 features from the key identifiers of Dalvik bytecode from each APK including word size, entropy, number of nodes and more. The authors of Androdet relied on a modular online learning algorithm to detect those three types, each separately, and their accuracies were 92.02%, 81.41% and 68.32% respectively. If a given APK is obfuscated with more than one of those algorithms, then the accuracy drops to 80.66%

Conti *et.al.* [15] relied on extending Androdet to improve detection accuracy and also added two new detection approaches. The first one is using Natural Language Processing to identify obfuscation from the code and the second is transforming the code into image and classify then using image recognition. In their paper, they tried each approach separately then tried to combine different approaches to figure out the one with best accuracy their approach reaches F1 score with **98.5%.**

Wang *et.al.* [16] proposed a different methodology as it identifies which obfuscator is used, not which technique is used as proposed in the literature. The authors collected and built their dataset from F-droid and started to obfuscate those files using different obfuscators as ProGuard, dashO, Allatori, Legu and Bangcle. They used different ML models and SVM achieved the highest accuracy of **97%**.

Table 2 provides a descriptive summary about previous work done in the field of android obfuscation detection using Machine Learning. The proposed work in [11] reaches the best accuracy among all of them which used Graph Convolutional Networks with Long Short-Term Memory (GCN-LSTM). This is a hybrid neural network architecture that combines two powerful techniques in the deep learning world While the field of Android obfuscation has seen significant development over the years, there remains a notable gap in the research landscape. Specifically, there is a limited number of studies addressing the challenges posed when multiple obfuscation techniques are combined together in the same sample. Next section will delve deeply into the methodology employed in our research on Android obfuscation. It will provide a comprehensive insight into the processes, techniques, and approaches used in the proposed method.

4. System Model and Evaluation

This section presents our proposed approach in detecting Android obfuscation with different obfuscation techniques. The proposed approach aims to detect android obfuscation that is obfuscated by more than one algorithm, mainly tested on renaming identifier and string encryption techniques both together as they are the main obfuscation techniques used in most cases. Figure 3 shows the proposed system model and will be discussed in details below.

Table 2: Summary of Related Work on Android Obfuscation.
Where: IR: Identifier Renaming, SE: String Encryption, CE: Class Encryption, CFG: Control Flow Graph, Acc: Accuracy.

Year	Obfuscator	Obfuscation Techniques	Approach	Measure
2022 [15]	obfuscAPK	IR, SE, Reflection, CE	NLP, Code to Image CNN, different ML models	Acc: 98.5%
2021 [11]	obfuscAPK	IR, SE, CFG	GCN-LSTM	F1: 98.95%
2019 [13]	Allatori	IR, SE, CFG	SVM, MOA	IR Acc: 98.5% SE Acc: 81.41% CFG Acc: 68.32% Overall Acc: 80.66%
2019 [12]	ProGuard dashO DexProtector DexGuard	IR, SE, Reflection, CFG	Random Forest, AdaBoost, Extra Trees, Linear SVM	Extra Trees: IR Acc: 70.1% SE Acc: 77% CFG Acc: 80% Ref Acc: 76.3% Avg Acc: 75.9%
2017 [16]	ProGuard, dashO, Allatori, Legu, Bangcle	IR, SE, CFG package modification	SVM, KNN, DT, MLP, AdaBoost, NB, logistic regression, RF	SVM Avg Acc: 97%

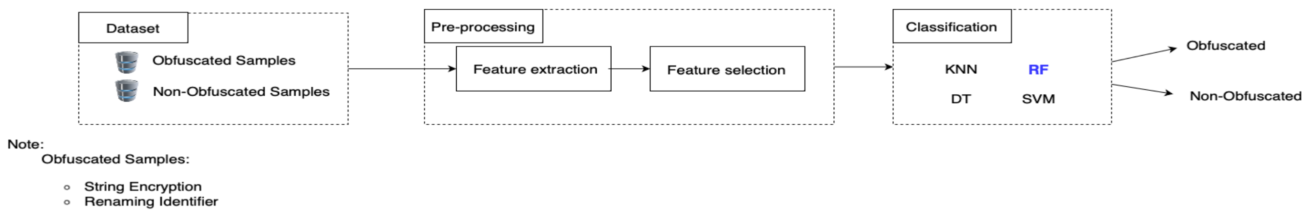


Figure. 3: proposed system model

4.1. Dataset

In the proposed Implementation, two distinct datasets were used: dataset (1) and dataset (2). Dataset (1) is for constructing and refining the model's parameters and it was split into training and testing set with a ratio of 70:30 and dataset (2) is for measuring the performance of the model on samples that mimic real-world APKs. The dataset is formed from malicious obfuscated and benign obfuscated and non-obfuscated samples. Malicious obfuscated samples are taken from [17], a new dataset that contains malware samples varies from 2018 to 2020 that were obfuscated using ObfuscAPK tool [11]. On the other hand, benign dataset was obtained by crawling F-Droid [18] and obfuscating the samples using ObfuscAPK. ObfuscAPK is an open-source tool that offers several obfuscation techniques options. When obfuscating benign samples, a mix of Trivial, Rename and Encryption options were used as shown below:

"*python -m obfuscapk.cli -o ClassRename -o FieldRename -o MethodRename -o ConstStringEncryption -o ResStringEncryption -o Reflection -o Rebuild -o NewAlignment -o NewSignature 'APK FILES' -p*"

Table 3: Descriptive Information about Dataset

	Dataset (1)	Dataset (2)
Obfuscated Benign	1801	-
Obfuscated Malicious	200	801
Non-Obfuscated Benign	1999	199

4.2. Preprocessing

Given an APK file, the features are extracted from classes.dex file. The features used in Androdet are used alongside with adding feature selection techniques relying on feature importance [15]. Feature importance is a technique were each feature is given a score based on its importance and these scores help in sorting the features. Figure 4 shows the rank of each feature from Androdet features for detecting both renaming identifier and string encryption.

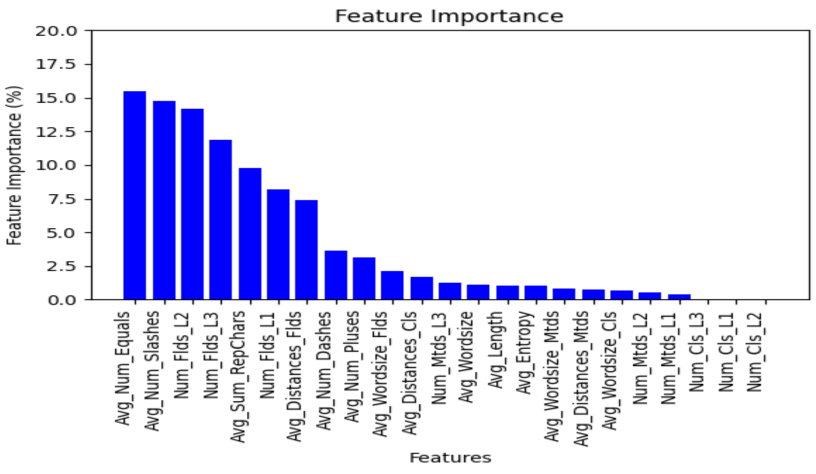


Figure. 4: Feature Importance Rank

As shown in Figure 4, the main features that affect the accuracy of the model are the average count of equals ("="), average count of existence of slashes ("/"), the number of fields with length 2 and 3, and Avg sum of repetitive characters which are appear more than once in a string. Upon evaluation, some features were found to have a detrimental effect on the model's accuracy. Consequently, these features were prudently removed to improve the overall performance and predictive capabilities of the model as Num_Cls_L1, Num_Cls_L2, Num_Cls_L3.

4.3. Evaluation Metrics

Evaluation metrics provide insights into how well the model is performing and help in comparing different models. Confusion matrix is a fundamental tool used in the evaluation of the performance of machine learning model as it assesses the accuracy and reliability of the model. Confusion matrix have four main components:

- True positive [TP]: The number of obfuscated APKs that were correctly predicted to be obfuscated
- True negative [TN]: The number non-obfuscated APKs that were correctly predicted to be non-obfuscated
- False positive [FP]: The number of obfuscated APKs but were wrongly predicted to be non-obfuscated.
- False negative [FN]: The number of non-obfuscated APKs but were wrongly predicted to be obfuscated.

Above components help in calculating evaluation matrix and to find accuracy, F1 score, precision and recall which can be calculated as follows:

- Accuracy: $\frac{Tp+Tn}{Tp+Fp+Tn+Fn}$
- Precision: $\frac{Tp}{Tp+Fp}$ Recall: $\frac{Tp}{Tp+Fn}$
- F1 score: 2 x $\frac{\Pr ecision \cdot Recall}{\Pr ecision + Recall}$

4.4. Results

This section compares the accuracy of different machine learning algorithms and presents the evaluation results to figure out the best performing algorithm to detect Android obfuscation. Four algorithms were chosen in this comparison: Random Forest (RF), Support Vector Machines (SVM), k-Nearest Neighbour (KNN) twice, first the N is set to 1 to makes the model more sensitive to local patterns and then the N is set to 5 for more stability and generalizable predictions. and Decision Tree (DT). Random Forest was the one with the best results as shown in Table 4.

Table 4: Machine Learning Models results with Dataset (1)

Model	Accuracy	F1	Precision	Recall
RF	**99.5%**	**99.5%**	**99.8%**	**99.1%**
KNN (k=5) KNN(k=1)	96.1% 95.8%	96.1% 95.8%	94.9% 94.8%	97.3% 96.7%
SVM	98.9%	98.9%	99.0%	98.8%
DT	98.2%	98.2%	98.2%	98.2%

To further analyze the results, the best model shown in Table 4, which is Random Forest, was evaluated against the different features explained in section 4.1. The experiment started by using only the most significant Feature, adding one feature at a time to see the best accuracy with the least number of features. As shown in Figure 5, the performance of the model stopped improving after adding the first 8 features together which means that other features can be removed to improve model performance and for better model interpretability. Table 5 shows the corresponding features that appear as numbers in Figure 5.

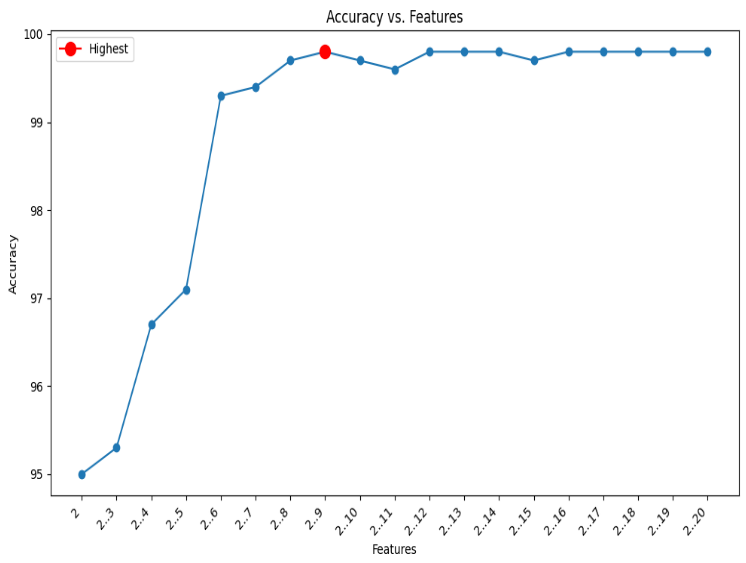


Figure. 5: RF Performance against Different Number of Features based on Feature Importance

Table 5: Features

No.	Feature	No	Feature
1	Avg_Num_Equals	10	Avg_Wordsize_Flds
2	Avg_Num_Slashes	11	Avg_Distances_Cls
3	Num_Flds_L2	12	Num_Mtds_L3
4	Num_Flds_L3	13	Avg_Wordsize
5	Avg_Sum_RepChars	14	Avg_Length
6	Num_Flds_L1	15	Avg_Entropy
7	Avg_Distances_Flds	16	Avg_Wordsize_Mtds
8	Avg_Num_Dashes	17	Avg_Distances_Mtds
9	Avg_Num_Pluses	18	Avg_Wordsize_Cls

Furthermore, after training and testing, the model was further evaluated using an additional dataset (Dataset (2)). Importantly, this dataset was entirely separate from the one used for training and testing our model. This separation between training and evaluation datasets is essential to ensure that our model's performance is tested on unseen files. Dataset (2) was tested with Random Forest model and gives an accuracy of 96.4%.

5. Conclusion and Future Work

Obfuscation is a very important technique that helps developers in protecting their Android applications and preventing them from any illegitimate usage. It also helps malware attackers in hiding their malicious code from being detected. Therefore, detecting obfuscation has become inevitable. This paper proposed several machine learning models that were used in detecting obfuscation and compare

between those models. We used Random Forest, SVM, KNN and DT. And random forest was the one with accuracy 99.5%. Not all types of obfuscation were covered in this paper.
This work paves the way in the process of detecting android malware as detecting obfuscation will help analysts in choosing the right technique in detecting malware and finding out that static analysis would be useless in those cases where APK is obfuscated.

References

1. S. G. Stats, “Operating system market share worldwide,” 2023. Available: https://gs.statcounter.com/os-market-share Accessed: (16 June 2023).
2. Balakrishnan, Arini, and Chloe Schulze. "Code obfuscation literature survey." CS701 Construction of compilers 19 (2005): 31.
3. Arnatovich, Yauhen & Wang, Lipo & Ngo, Ngoc & Soh, Charlie. (2018). A Comparison of Android Reverse Engineering Tools via Program Behaviors Validation Based on Intermediate Languages Transformation. IEEE Access. PP. 1-1. 10.1109/ACCESS.2018.2808340.
4. Apktool. https://ibotpeaches.github.io/Apktool. (Accessed 10 September 2023).
5. Dex2jar. https://bitbucket.org/pxb1988/dex2jar. (Accessed 10 September 2023).
6. Proguard. https://www.guardsquare.com/proguard. (Accessed 3 July 2023).
7. Dexguard https://www.guardsquare.com/dexguard. (Accessed 3 July 2023).
8. Allatori. https://allatori.com/overview.html. (Accessed 3 July 2023).
9. DashO. https://www.preemptive.com/products/dasho/. (Accessed 3 July 2023).
10. S. Aonzo, G. C. Georgiu, L. Verderame, and A. Merlo, “Obfuscapk: An open-source black-box obfuscation tool for android apps,” SoftwareX, vol. 11, p. 100403, 2020.
11. S. Jiang, Y. Hong, C. Fu, Y. Qian, and L. Han, “Function-level obfuscation detection method based on graph convolutional networks,” Journal of Information Security and Applications, vol. 61, p. 102953, 2021
12. M. Park, G. You, S. Cho, M. Park, and S. Han, “A framework for identifying obfuscation techniques applied to android apps using machine learning,” J. Wirel. Mob. Networks Ubiquitous Comput. Dependable Appl., vol. 10, no. 4, pp. 22–30,2019.
13. O. Mirzaei, J. de Fuentes, J. Tapiador, and L. Gonzalez-Manzano, “Androdet: An adaptive android obfuscation detector,” Future Generation Computer Systems, vol. 90, pp. 240–261, 2019.
14. Zien, Alexander, et al. "The feature importance ranking measure." Machine Learning and Knowledge Discovery in Databases: European Conference, ECML PKDD 2009, Bled, Slovenia, September 7-11, 2009, Proceedings, Part II 20. Springer Berlin Heidelberg, 2009.
15. M. Conti, V. P., and A. Vitella, “Obfuscation detection in android applications using deep learning,” Journal of Information Security and Applications, vol. 70, p. 103311, 2022.
16. Y. Wang and A. Rountev, “Who changed you? obfuscator identification for android,” in 2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft), pp. 154–164, IEEE, 2017.
17. S. Kumar, D. Mishra, B. Panda, and S. K. Shukla, “Androobfs: time-tagged obfuscated android malware dataset with family information,” in Proceedings of the 19th International Conference on Mining Software Repositories, pp. 454–458, 2022.
18. F-Droid. https://f-droid.org/. (Accessed 14 September 2023).
19. Chen, Haibo, et al. "Control flow obfuscation with information flow tracking." Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture. 2009.

20. Faisal, H., Hindy, H., Gaber, S. and Salem, A.B., 2022, December. ASurvey ON ARTIFICIAL INTELLIGENCE TECHNIQUES FOR MALWARE DETECTION. In CS & IT Conference Proceedings (Vol. 12, No. 23). CS & IT Conference Proceedings.