DETECTING ANDROID OBFUSCATION METHODS WITH LSTM

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF INFORMATICS OF
THE MIDDLE EAST TECHNICAL UNIVERSITY

BY

BULUT ULUKAPİ

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE
OF
MASTER OF SCIENCE
IN
THE DEPARTMENT OF CYBER SECURITY

JUNE 2022

Approval of the thesis:

DETECTING ANDROID OBFUSCATION METHODS WITH LSTM

submitted by **BULUT ULUKAPİ** in partial fulfillment of the requirements for the degree of **Master of Science in Cyber Security Department, Middle East Technical University** by,

_____

_____

_____

_____

_____

_____

_____

Date: 17.06.2022

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Surname: **Bulut Ulukapi**

Signature :

ABSTRACT

DETECTING ANDROID OBFUSCATION METHODS WITH LSTM

JUNE 2022, 77 pages

In software development, obfuscation means intentionally designing the source code to make it more difficult to understand by humans, thus making the analysis and reverse engineering challenging to accomplish. Obfuscation methods modify the source code syntactically while maintaining its original functionality. Obfuscation is also integrated into the Android development environment, and it has been used widely by legitimate software developers and malware authors. The app developers employ obfuscation methods to protect intellectual property rights, whereas malware authors use them for evading detection and digital forensics. The widespread usage of obfuscation techniques presents challenges to researchers and analysts who focus on app cloning, repackaging, third-party library, and malware detection. Hence, it is crucial to detect obfuscation for building reliable, effective, and automated detection systems. In the present study, we utilized Natural Language Processing (NLP) techniques and Long-short Term Memory (LSTM) to detect whether a given application is obfuscated or not. For this, we collected applications from F-Droid, an open-source Android app repository, and obfuscated them with nine different obfuscation techniques. We performed experiments and obtained promising results for detecting different obfuscation methods. We also observed that the model is more successful in detecting specific obfuscation methods than the others.

Keywords: Android obfuscation, Long-Short Term Memory, Natural Language Processing, obfuscation detection

ÖZ

LSTM İLE ANDROID GİZLEME YÖNTEMLERİNİN TESPİTİ

Haziran 2022, 77 sayfa

Yazılım geliştirmede, gizleme, kaynak kodunun insanlar tarafından anlaşılmasını zorlaştıracak şekilde kasıtlı olarak tasarlanmasıyla analiz ve tersine mühendisliğin başarılmasının zor hale getirilmesi anlamına gelir. Gizleme yöntemleri, programın orijinal işlevselliğini korurken kaynak kodunu sözdizimsel olarak değiştirir. Gizleme, Android geliştirme ortamına da entegre edilmiştir ve hem yazılım hem de kötü amaçlı yazılım geliştiricileri tarafından yaygın olarak kullanılmaktadır. Uygulama geliştiricileri fikri mülkiyet haklarını korumak için gizleme yöntemleri kullanırken, kötü amaçlı yazılım geliştiricileri bunları tespitten ve dijital adli tıptan kaçmak için kullanır. Gizleme tekniklerinin yaygın kullanımı, uygulama klonlama, yeniden paketleme, üçüncü parti kitaplık ve kötü amaçlı yazılım tespit etmeye odaklanan araştırmacılar ve analistler için zorluklar sunar. Bu nedenle, güvenilir, etkili ve otomatik algılama sistemleri oluşturmak için gizleme yöntemlerini tespit etmek çok önemlidir. Bu çalışmada, belirli bir uygulamanın bulanık olup olmadığını tespit etmek için Doğal Dil İşleme (NLP) teknikleri ve Uzun Kısa Süreli Bellek (LSTM) kullandık. Bunun için açık kaynaklı bir Android uygulama deposu olan F-Droid'den uygulamalar topladık ve bunları dokuz farklı gizleme tekniği kullanarak gizledik. Farklı gizleme yöntemlerini tespit etmek için deneyler yaptık ve umut verici sonuçlar elde ettik. Ayrıca modelin belirli gizleme yöntemlerini tespit etmede diğerlerinden daha başarılı olduğunu gözlemledik.

Anahtar Kelimeler: Android kod gizleme yöntemleri, Uzun-kısa Dönem Hafıza, Doğal Dil İşleme, Gizleme Yöntemleri Tespiti



To My Mother

ACKNOWLEDGMENTS

There are several people I am grateful for their help during my thesis process. First of all, I would like to thank my supervisor Assoc. Prof. Dr. Cihangir Tezcan for his support and guidance. I also would like to thank my co-advisor, Dr. Pınar Gürkan Balkıçlıoğlu, for her inspirational ideas and contributions to this thesis.

Moreover, I would like to thank Assoc. Prof. Dr. Cengiz Acartürk for his mentoring, support, and motivation from the very beginning throughout my master's and thesis writing process. I would also like to thank the members of our malware analysis research group, especially Melih Sırlancı, for his help and support in the scope of this study.

I would like to thank the Scientific and Technological Research Council of Turkey (TUBITAK) and TUBITAK BIDEB for providing me with the General Domestic Master's Scholarship BIDEB 2210-A. Besides, the numerical calculations reported in this paper were partially performed at TUBITAK ULAKBIM, High Performance and Grid Computing Center (TRUBA resources).

Lastly and most importantly, I would like to thank my mother, Seval, my sisters Başak and Duygu, and all my family and friends. I truly appreciate their love, support, and encouragement every day. Their presence is invaluable to me in every moment of my life.

TABLE OF CONTENTS

ABSTRACT iv

ÖZ v

DEDICATION vi

ACKNOWLEDGMENTS vii

TABLE OF CONTENTS viii

LIST OF TABLES xi

LIST OF FIGURES xii

LIST OF ABBREVIATIONS xiv

CHAPTERS

1 INTRODUCTION 1

1.1 Motivation and Problem Definition 3

1.2 Research Questions 4

1.3 Organization of the Thesis 4

2 BACKGROUND AND RELATED WORK 5

2.1 Android Architecture 5

2.1.1 Android Application Package 5

2.1.2 Dalvik Bytecode and Smali Language 7

2.2 Obfuscation 7
2.2.1 Android Obfuscation Techniques 10
2.2.1.1 Trivial Techniques 11
2.2.1.2 Non-Trivial Techniques 14
2.2.1.3 Preventive Techniques 18
2.2.1.4 Custom Techniques 18
2.2.2 Android Obfuscation Tools 19
2.3 Signature-based Methodologies 21
2.4 Machine Learning Methodologies 23
2.5 Deep Learning Methodologies 25
2.5.1 Neural Networks 25
2.5.2 Natural Language Processing 29
2.5.3 Detection Methodologies with Deep Learning and NLP 30
2.6 Obfuscation Detection Techniques 31
2.7 Summary 33
3 METHODOLOGY 35
3.1 Scope and Approach 35
3.2 Overview of the Proposed Methodology 37
3.3 Datasets 39
3.4 LSTM model 46
3.4.1 Environment Setup 47
3.4.2 Training/Testing 49
3.5 Summary 52

4 RESULTS 53

4.1 Experimental Results 53

4.2 Discussion 54

5 CONCLUSION 59

5.1 Conclusion 59

5.2 Limitations and Future Work 60

REFERENCES 63

APPENDICES

A CONFUSION MATRICES 73

LIST OF TABLES

TABLES

Table 3.1 The characteristics of the two datasets (ISD and MSD) 46

Table 3.2 The used libraries and their versions to build the LSTM model . . . 47

Table 4.1 The results obtained from the models trained by the Instruction Sequence Dataset (ISD) and the Method Sequence Dataset (MSD) 54

LIST OF FIGURES

FIGURES

Figure 2.1 The structure of an Android Application Package 6

Figure 2.2 How to build and run Android apps 8

Figure 2.3 Android Obfuscation Techniques 12

Figure 2.4 The architecture of an LSTM cell [1] 27

Figure 3.1 The overview of the methodology in the present study 38

Figure 3.2 The processing pipeline for each detection unit in the model . . . 39

Figure 3.3 A sample code before identifier renaming obfuscation 40

Figure 3.4 A sample code after identifier renaming obfuscation 40

Figure 3.5 A sample code before constant string encryption 41

Figure 3.6 A sample code after constant string encryption 41

Figure 3.7 Added code in arithmetic branch insertion 42

Figure 3.8 Added proxy methods in call indirection 42

Figure 3.9 A sample code after reorder 43

Figure 3.10 Added code in goto insertion 43

Figure 3.11 Added code in method overload 44

Figure 3.12 Sample lines from the ISD 45

Figure 3.13 Sample lines from the MSD 45

Figure 3.14 The layers of our proposed model 51

Figure 4.1 Confusion matrix for reference 53

Figure A.1 Confusion Matrices for Identifier Renaming 73

Figure A.2 Confusion Matrices for Constant String Encryption 74

Figure A.3 Confusion Matrices for Arithmetic Branch Insertion 74

Figure A.4 Confusion Matrices for Call Indirection 75

Figure A.5 Confusion Matrices for Reflection 75

Figure A.6 Confusion Matrices for Reorder 76

Figure A.7 Confusion Matrices for Nop Insertion 76

Figure A.8 Confusion Matrices for Goto Insertion 77

Figure A.9 Confusion Matrices for Method Overload 77

LIST OF ABBREVIATIONS

ML	Machine Learning
DL	Deep Learning
NLP	Natural Language Processing
RNN	Recurrent Neural Network
LSTM	Long Short-Term Memory
ISD	Instruction Sequence Dataset
MSD	Method Sequence Dataset
APK	Android Application Package
JVM	Java Virtual Machine
DVM	Dalvik Virtual Machine
API	Application Programming Interface
ANN	Artifical Neural Network
MAIL	Malware Analysis and Intermediate Language
CFG	Control Flow Graph
SVM	Support Vector Machine
kNN	k-Nearest Neighbor
CNN	Convolutional Neural Networks
BPTT	Backpropagation Through Time
GRU	Gated Recurrent Network
MLP	Multilayer Perceptron
DBN	Deep Belief Networks

CHAPTER 1

INTRODUCTION

Today, with the developments in Information and Communication Technologies, the convenience and benefits of mobile devices have made them an inseparable part of our daily lives. Mobile devices provide many services and functionalities, including instant messaging, banking, navigation, entertainment, and healthcare. They are used by many people due to these services and conveniences they provide. On the other hand, the Android operating system is the most preferred choice for mobile devices and has a market share of 71% for smartphones and 44% for tablets [2]. Due to the popularity of Android, there are numerous applications in various app stores, as well as a significant number of malware that target Android platforms. For example, Google Play Store alone has 2.6 million applications available to download in December 2021 [3]. Also, it is reported that more than 2 million malware packages were installed on Android mobile devices in the first half of 2021 [4]. The widespread usage of Android, including applications that contain sensitive, private or financial information, creates various attack surfaces for malicious actors. At the same time, these dangerous attack surfaces also attract the research community to build analysis methods and mechanisms for securing Android applications.

Due to the nature of the Java programming language and development environment, Android applications are easy to disassemble and decompile, making them vulnerable to reverse engineering. Reverse engineering refers to the process of dissecting an object or an executable file to understand and disclose its design logic and functioning. An Android application can be easily analyzed by reverse engineering and even rebuilt to have the same functionality or with some modifications. The fact that Android applications can be easily reverse engineered makes it easier to analyze malware and take the necessary measures to mitigate it. However, it also allows malicious actors to decode the business logic of many applications and copy them for various purposes, such as stealing the intellectual property of other companies or concealing malicious payloads into popular applications and redistributing them in different less secure app stores. Therefore, there is a need to secure Android applications and their source code for both software and malware developers.

Obfuscation is one of the most often employed Android application protection mechanisms. In general terms, obfuscation modifies a program's source code, making it more difficult to understand by human analysts. Various obfuscation techniques can be used for different purposes. Software developers utilize obfuscation to protect their intellectual property, such as proprietary/business algorithms, and prevent end-users from obtaining secret information from the program's source code. Furthermore, developers use obfuscation for optimization purposes [5]. On the other hand,

implementing obfuscation methods is also popular among malicious actors. Malware authors implement different and sometimes even custom techniques to create different-looking versions of their malware to evade detection engines and conceal malicious payloads. Besides, obfuscation enables malware authors to hide critical information and their tracks in the source code to make the reversing and analysis much harder for digital forensics. Previous research has investigated obfuscation usage in the wild and showed that obfuscation is frequently used in common app repositories. In [6], after examining 1.7 million Google Play Store applications, researchers stated that 25% of the investigated apps are obfuscated. The authors also state that the percentage of the obfuscated apps increases to 50% for the most popular apps. In [7], researchers investigated both benign and malware samples. They reported that malware developers put more effort into the obfuscation process and use more custom and advanced techniques. Also, the usage of string encryption is more prevalent in malware samples than benign samples.

The widespread use of obfuscation techniques in Android applications affects various research domains. Hence, the previous research has studied various aspects of obfuscation, such as theoretical perspectives of code obfuscation and obfuscation algorithms [8, 9], clone detection [10, 11], repackaging detection [12, 13], effects of obfuscation [14], packing detection [15], detecting obfuscated malware [16, 17], and detecting third-party libraries [18,19]. Due to the effects of obfuscation techniques on other research domains and detection systems, it becomes crucial to detect whether a given Android application is obfuscated or not. Furthermore, it is also important to identify the applied obfuscation techniques if the application is obfuscated. Detecting obfuscation in Android provides various insights, which will be detailed under the Motivation and Problem Definition section, into clone detection, third-party library detection, malware detection, and digital forensics to build efficient and reliable systems.

Obfuscation detection methods have similar historical developments and trends as other detection strategies such as clone and malware detection. For example, malware detection studies utilized signature-based techniques in the beginning. In these methods, signatures are created from specific malware patterns and strings. Then, the detection is performed by comparing the input and available signatures [20,21]. Nevertheless, signature-based methods became obsolete since it is much easier to change the source code and evade detection than to create a signature. With the emergence of machine learning models, the detection paradigm shifted towards machine learning (ML) based detection methods to overcome the limitations of signature-based detection systems. ML-based methods performed better than signature-based methods; however, they still require the extraction of features through careful human analysis of specific applications. The intense effort for feature engineering became an obstacle to automated, reliable, and generalizable detection [22, 23]. More recently, deep learning approaches, especially Recurrent Neural Networks (RNN) and Long Short-Term Memory (LSTM) models, provided solutions to overcome this problem [24–26]. Many studies focusing on malware detection problems have also started to employ natural language processing (NLP) techniques based on the similarities between natural languages and programming languages [26]. Obfuscation detection

methodologies in Android also started with signature-based methods. Then, the research community incorporated ML and then DL models to build more robust and automated systems for detecting obfuscation techniques similar to malware detection.

In the present study, we proposed an NLP-based approach for obfuscation detection in Android by utilizing a particular type of Recurrent Neural Network (RNN), namely Long Short-Term Memory (LSTM). Since it is difficult to establish or find a dataset that covers virtually all the available obfuscation methods, we chose nine popular obfuscation techniques and created our obfuscated samples. Then we disassembled all the applications, including original and obfuscated samples, statically and obtained the Smali codes. From the Smali codes, we represented applications in two forms and created the instruction sequence dataset (ISD) and the method sequence dataset (MSD), similar to the study in [26]. We created these datasets for every obfuscation method separately. Then we performed binary classification for each obfuscation technique. We utilized an LSTM model for the classification and trained the model with the ISD and MSD separately. The proposed models achieved higher detection accuracies for specific obfuscation methods than the others.

1.1 Motivation and Problem Definition

Obfuscation is a popular and effective technique to protect applications against reverse engineering in the Android environment. However, it introduces many challenges in numerous research areas, including malware, clone, and third-party library detection. For example, in the malware detection context, obfuscation can be used to evade detection engines and complicate malware analysis. Thus, detecting obfuscation is an essential step in revealing the behavior of malware. It also provides insights for digital forensics experts into the tools and techniques that malicious actors employ to evade detection. On the other hand, it is also crucial to detect obfuscation in the third-party library detection context. Since third-party libraries may contain many security vulnerabilities that can be used as a foothold to infiltrate mobile devices, it is vital to identify them correctly. To sum up, detecting obfuscation is a fundamental problem for enhancing the security of mobile devices in general. Thus, in this thesis, we focused on the problem of detecting obfuscation methods in the Android environment.

NLP techniques have been employed for a while in various detection areas, especially malware detection. It is shown that utilizing NLP methods such as text classification and language modeling is effective for detecting malware [26]. Moreover, the malware detection community has embraced DL models and methodologies to overcome the limitations of feature engineering. Previous studies show that the LSTM model can achieve high detection rates for detecting malware [24–26]. Obfuscation detection methods show parallelism to malware detection studies. Besides, obfuscated codes have some structural characteristics, repetitive patterns, and long dependencies. Considering LSTM's success in the malware detection area and its effectiveness in dealing with vanishing and exploding gradient problems, LSTM architecture is also a suitable candidate for obfuscation detection. LSTM can effectively extract semantic information from long sequential data [26–28] . Therefore, in this study, we utilized

NLP techniques, specifically language modeling using neural networks and the LSTM model, and wanted to investigate their performance in detecting obfuscation.

1.2 Research Questions

The research questions of this thesis can be stated as follows. First, we wanted to examine if it is feasible to treat the source code of Android applications as natural languages and employ NLP techniques in the context of obfuscation detection. More specifically, we wanted to investigate if it is a sensible approach to model disassembled smali language of original and obfuscated Android applications to classify the code pieces of applications as natural languages. Furthermore, the current work explores instruction and method as the two representation units in the smali language and build language models based on these two units to determine which is better suited for grasping more semantic information to detect obfuscation using the smali code of applications. Lastly, we also wanted to see the performance of the LSTM model in detecting different obfuscation methods.

1.3 Organization of the Thesis

This thesis is organized as follows. In Section 2, we will provide the necessary background information related to the scope of this study. We will also present the relevant work on obfuscation in Android and detection methodologies, including malware, clone, third-party library, and obfuscation. Section 3 will describe the scope, approach, and overview of our methodology. We will describe which obfuscation techniques we applied and how we created our datasets for the study. Then we will give the details of our LSTM model, including the setup of the environment, training/testing, parameters, and configuration. Then, we will present the study results and a discussion of the results. Lastly, we will conclude the thesis and state the limitations of the present study and the future work in Section 5.

CHAPTER 2

BACKGROUND AND RELATED WORK

This section will introduce the concepts and techniques related to the study. We will start by explaining the structure of an apk file and the relation between Dalvik bytecode and the smali language. Then we will describe what the obfuscation is and continue with the varying obfuscation methods and tools in the Android environment. We will also present detection methodologies in general and state their changing trends. After giving the relevant details on neural networks and Natural Language Processing, we will conclude this section by reviewing the literature on Android obfuscation detection.

2.1 Android Architecture

2.1.1 Android Application Package

Android applications are primarily written using Kotlin, Java, and C++ languages. The app code is compiled with any other data and resource files into an APK (Android application package) file to build the application. Hence, an Android application is essentially a compressed archive file with the .apk extension. The apk file of an application includes the contents required at the runtime and is necessary to install the application on Android-powered devices [29]. The structure of an apk file is in Figure 2.1.

The compressed apk file consists of four directories (res, assets, lib, and META-INF) and three files (AndroidManifest.xml, classes.dex, and resources.arsc). The functions of these directories and files are as follows [7, 29, 30]:

META-INF/: This folder contains various files related to the manifest information and any other metadata about the Android package. For example, MANIFEST.MF file includes the necessary information for the run-time environment as well as build number, security permissions, creator, and version of the package. CERT.SF and CERT.RSA files include the signature information of the application. These files are used to sign the contents of the application. They can also be used to check and verify the apk file's integrity.

assets/: This directory stores the external resources used in the application, such as audio files, images, videos, fonts, XML, and HTML. The assets directory allows developers to construct subdirectories at any depth with customizable file structures

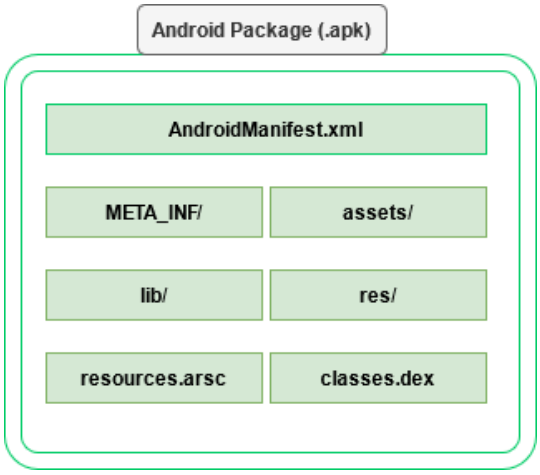


Figure 2.1: The structure of an Android Application Package

and arbitrary files. This feature also allows malware authors to place executables containing exploits under the assets directory.

res/: This directory is similar to assets/ directory. It stores Android resource files that will be mapped into the .R file, such as bitmaps, layout definitions, animations, and user interface strings. Still, there are some differences between res/ and assets/ directory. While assets/ directory allows flexible file names and storing files in subdirectories, res/ directory gives pre-compiled IDs to each file for easy and quick access. Res/ directory also ensures correct file name by compile-time checking, but it is more restricted than assets/ directory from the developer's perspective.

lib/: This folder contains the library files for specific platforms. Subdirectories of the lib directory can be created according to the type of the processors used on the device, such as armeabi, misp, and x86.

AndroidManifest.xml: In the root directory of every application, there must be an AndroidManifest.xml file with that exact name. This file declares the critical information about the application to the Android system. The Android system needs the information provided by the manifest file beforehand to be able to run any of the application's code. Basically, this XML file can be seen as the configuration of an application. It contains the description of the application's components and basic information such as version, name, required permissions, and intents of an apk. Therefore, AndroidManifest.xml is a widely studied and analyzed part of an Android package by analysts and researchers to obtain the behavior of an application.

resources.arsc: This file records and tracks the relationship information between the resource files and their related resource IDs. It can be used to locate specific resources quickly.

classes.dex: This file is a Dalvik Virtual Machine (DVM) executable that contains all the information of all the classes used in the app. Essentially, the data in this file is organized in a way that the DVM can understand and execute. DVM is a virtual

machine similar to Java Virtual Machine (JVM) and is optimized for mobile devices and systems with low resources in general. Classes.dex file can be disassembled into the .smali files using a disassembler so that the application's source code can be obtained. Hence for topics such as malware detection, clone detection, and third-party library detection, this file has significant importance and is focused on by developers, analysts, and researchers. This file is also one of the most targeted parts of an Android package for obfuscation. Therefore, we also focused on classes.dex file in our study and obtained the smali representation of the Dalvik bytecode for obfuscation detection. The following section will detail JVM, DVM, and the Dalvik bytecode's relation with the smali language.

2.1.2 Dalvik Bytecode and Smali Language

Java Virtual Machine (JVM) is a stack-based virtual machine that allows java code to run on different platforms. JVM can be considered as an abstract layer between the program and the operating system on which the code is running. When a program is developed in Java, it gets converted into the platform-independent Java bytecode format by the Java compiler. In the case of Android, JVM is replaced with Dalvik Virtual Machine (DVM). DVM is a register-based virtual machine that runs Android applications. Due to the limited resources of Android-powered mobile devices, DVM is optimized to run smoothly even on low processing power and memory. We noted that even though Android applications are primarily written in Java, languages like Kotlin, C++, C, and Javascript can be used to develop applications. Whatever the chosen language is, the source code of the applications first gets converted to the Java bytecode (.class file). Then, the Dex compiler converts Java bytecode into Dalvik bytecode (.dex file) so that DVM can understand and execute the Android application. Figure 2.2 below shows how an Android application is built and run.

As we mentioned in the previous section, we focused on classes.dex file of apks which contains the Dalvik bytecode of the application in binary. Dalvik bytecode is the format that the machine understands. Nonetheless, it is not easy to understand or modify (including obfuscation transformations) for developers. Therefore, bytecode needs to be transformed into Smali (assembly) language using disassemblers. Smali is known as the most common human-readable representation. It can be thought of as an equivalent of the assembly code of a C program. Various disassemblers can be used to obtain smali representations of Dalvik bytecode, such as apktool [31], JEB [32], baksmali [33], IDA Pro [34], Androguard [35], and dex2jar [36]. While reviewing the literature, we saw that the Dalvik bytecode and Smali language are being used interchangeably. In this study, when we refer to the Dalvik bytecode, we mean the binary bytecode in classes.dex file, and when we refer to the smali language, we mean the disassembled version of bytecode into a more human-readable format.

2.2 Obfuscation

There are several approaches to ensure the security of desired assets, information, and systems in cyber security. The standard and accepted approach is called security by

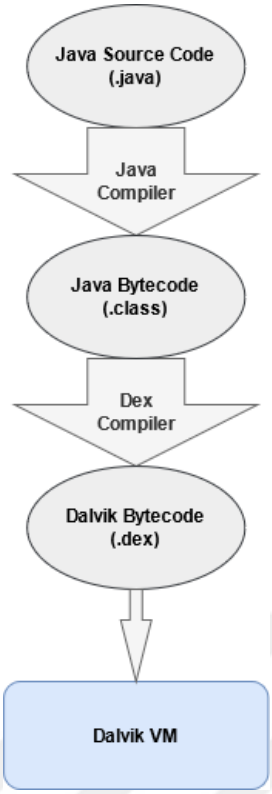


Figure 2.2: How to build and run Android apps

design. For example, in cryptosystems, this approach states that even though everything about a system is public knowledge, the system should be secure as long as the secret key is protected. There is also another approach called security through obscurity, and it is considered a bad practice in terms of security. In this approach, secrecy is the main component, and the approach mostly hides crucial information to provide security. Nevertheless, when used as an independent layer, it can be considered a useful, complementary tool for security. Xu et al. also state two similar approaches for obfuscation in their survey [37]. Researchers categorize obfuscation mainly under two approaches: Code-Oriented Obfuscation and Model-Oriented Obfuscation. Model-Oriented Obfuscation is investigated mainly by scientists who focus on theoretical studies on circuits and Turing machines as in study [38]. On the other hand, this approach also attracts cryptographers' attention since the obfuscation methods are primarily based on cryptographic primitives. The Model-Oriented approach aims to prevent an adversary from inferring the functioning of a computational model. Unlike the previous approach, Code-Oriented Obfuscation interests the research community of software security experts, engineers, and researchers to protect software, including malware detection research. The Code-Oriented approach aims to make the software harder to interpret and reverse. In this thesis, we focus on Code-Oriented obfuscation, and throughout the study, when we use the word obfuscation, we refer to the Code-Oriented approach.

Code obfuscation is an effective technique that makes the source code of a program unintelligible for humans to thwart reverse engineering attacks. In [39], Collberg et al. give the widely used and unified definition of obfuscation. They define a trans-

formation T that changes the code of a source program P into a target program P'. The input program P can be in the form of source code, machine code, or bytecode. The researchers call the transformation T an obfuscation transformation if both P and P' have the same observable behavior. They also state the following conditions in their study for an obfuscation modification to be a proper, valid transformation: if program P cannot terminate, or terminates because of an error, program P' may or may not terminate; otherwise, P' must terminate and give the same output as P. In other words, all kinds of modifications that change the physical appearance of the code while preserving its original semantics and functionality can be considered as obfuscation. Hence, there are many different obfuscation techniques with their own construction methodology and specific patterns. We should also note that, although obfuscation transformations keep the original semantics and produce the same observable behavior, they may still introduce some side-effects to the code, such as creating/transferring files and sending messages. As long as the user is not aware of the side effects, observable behavior does not change since it refers to the behavior perceived by the user.

Obfuscation can be considered as security through obscurity approach, which we explained previously. Hence, it has minimal security guarantees regarding cryptographic standards and cannot provide a security level as a proper implementation of encryption can. Studies [40] and [41] also showed that achieving perfect obfuscation is impossible due to the existence of functions that cannot be obfuscated, and obfuscation cannot provide virtual black-box property for the program. Therefore, it is safe to say that obfuscation does not aim to provide solid security for the program. Instead, it aims to make the analysis and understanding of the program more difficult by consuming more resources. The resources needed for reversing a program can be time, hardware, toolset, computing power, and brainpower. Obfuscation can also be thought of as a puzzle for analysts reading the code. Eventually, it can be solved by providing the required attention and effort, but the solving process still can be a tedious job for the analysts. Thus, obfuscation techniques are widely used in practice to protect the source code of programs against reverse engineering.

Like most of the concepts in cyber security, obfuscation is also a double-edged sword meaning it can be used both for legitimate and malicious purposes. In the early times, code obfuscation is mainly driven by the need for Digital Rights Management (DRM) and the protection of the intellectual property of commercial software. The goal of the obfuscation used for protecting intellectual property is to prevent end-users from obtaining secret information from the program's source code. Such secret information can be proprietary/business algorithms, ways to unlock paid program features, and cryptographic keys. Furthermore, obfuscation can also be used to optimize the programs and increase their performance. During the development process of a program, source code is written in a way that all of the developers working on the project can understand and contribute to the project. This approach means that the process is mainly human-oriented. However, the platform which will run the application does not need the aforementioned human-oriented approach. Therefore, developers can use obfuscation methods (e.g., changing the identifier names with shorter ones) to improve their program performance.

On the other hand, implementing obfuscation methods has also gained popularity among malware developers. Malicious actors implement different and sometimes

even custom techniques to create different-looking versions of the malicious code to evade detection [42]. Most antivirus scanners and detection engines are signature-based systems. To create a valid signature for detection, an analyst has to reverse the malware to find out its internal structures, algorithms, and behavior which is a laborious and time-consuming process. On the contrary, the malware developer can easily change the look of its code by implementing obfuscation and then evade signature-based detection systems. Moreover, malware authors leverage obfuscation to hide critical information and their tracks in the source code to make the reversing and analysis much harder for digital forensics. As we can see, there are various reasons to implement obfuscation in practice. Therefore, the research on the theoretical perspective of code obfuscation and obfuscation algorithms is quite well-studied, and countless new algorithms or implementations keep appearing [8, 43–45]. These advancements have also led to the development of many open-source and commercial obfuscation tools for different programming languages and operating systems, including C, C++, Java, Javascript languages, and Windows, macOS, Linux, and Android environments.

In [46], researchers categorize obfuscation transformations as static and dynamic transformations. Static transformations are applied to programs or their data once during development, compilation, linking, or update so that the programs or their data stay the same during execution. Dynamic transformations are applied similarly, but the difference is that the program and its data change during loading or execution. Although dynamic obfuscation transformations are more substantial against reverse engineering attacks, they bring more performance overhead since the code may need to be written before execution. The similarity between the development and categorization of obfuscation techniques and malware analysis methods once again shows us the arms race between defenders and attackers in the cyber security domain and how they have affected each other throughout history. In this study, we focused on the static obfuscation transformations on the Android platform by utilizing Smali representations of applications. Hence, we will explain the current state and trends of obfuscation in the Android environment and state the most common obfuscation methods and tools in the next section.

2.2.1 Android Obfuscation Techniques

In Section 2.1, we explained how Android applications are built. Due to the nature of Java, Android apps are more straightforward to disassemble and decompile than other binary codes, which makes them more vulnerable to reverse engineering attacks. Therefore, Android developers and researchers used obfuscation to make reversing an application harder for various reasons, including those mentioned in the previous section. Let us take a closer look at obfuscation in the Android environment. There are many Android applications available for users to download in different markets. According to the study [47], more than 25% of applications in the Google Play Store are clones of other applications. Moreover, the study [48] states that 80% of the most popular 50 applications have fake versions due to repackaging. Thus, there is a need to secure the Android applications against reverse engineering. Obfuscation is a popular and effective technique to secure the apps and protect the developers' and companies' intellectual property and commercial interests. Moreover, the usage of

obfuscation is also strongly recommended for all applications by the Android Developer Guide [49].

In [6], Wermke et al. investigated 1.7 million benign apps from Google Play Store to detect whether an obfuscation method is implemented in an app or not. They found out that 25% of the investigated apps are obfuscated. Researchers also state that the percentage of the obfuscated apps increases to 50% for the most popular apps, which are over 10 million downloads. This rise in percentage shows us that when the commercial concerns of an app increase, obfuscation becomes a popular solution. On the other side of the scale, malicious Android developers also employ various obfuscation techniques for different purposes. They can use obfuscation to evade signature-based malware detection engines, conceal malicious payload, and hide information that can be valuable for digital forensics analysis, such as the IP address of the CommandControl server. Malware authors can also hide their malicious code inside benign applications by repackaging. In [7], Dong et al. collected both benign and malicious Android applications from multiple third-party markets, Google Play, and malware databases to investigate the obfuscation techniques used in the wild. Researchers point out that the status of the obfuscation methods varies in many aspects for different applications. For example, malware developers put more effort into the obfuscation process and use more custom and advanced techniques. Also, the usage of string encryption is more prevalent in malware samples. Besides, researchers state that there are more packed apps in third-party markets than in Google Play. As we can see, there are various obfuscation methods and implementations in Android for numerous different reasons on different platforms. To give a holistic view of obfuscation on Android, we reviewed the literature and categorized the most common and widely used obfuscation techniques. We classified Android obfuscation methods under four categories as Trivial, Non-Trivial, Preventive, and Custom techniques. The classification can be seen in Figure 2.3.

2.2.1.1 Trivial Techniques

Trivial obfuscation techniques are generally simple transformations that require little technical knowledge and skill set. These techniques mainly consist of simple operations which do not necessitate code-level changes. In some cases, these methods are used to optimize the application's code. Malicious actors use trivial methods to evade detection engines that use basic signatures of the complete application or some specific part. Since the implemented modifications are basic operations, trivial techniques do not have real obfuscation effects on the code. However, they can still successfully evade some signature-based detection engines [30]. The most common obfuscation methods that fall under this category can be stated as follows:

- **Realignment**: To build an Android application, developers need to sign and align the application as the last two steps. zipalign, an alignment tool of the Android SDK, can be used to align signed applications. The tool optimizes the application to increase performance by mostly aligning uncompressed data such as media and raw files [50]. Aligning reorders and optimizes the file structure so that the apk can be mapped in memory more efficiently. This process creates a somewhat different file while preserving the original functioning. Therefore,

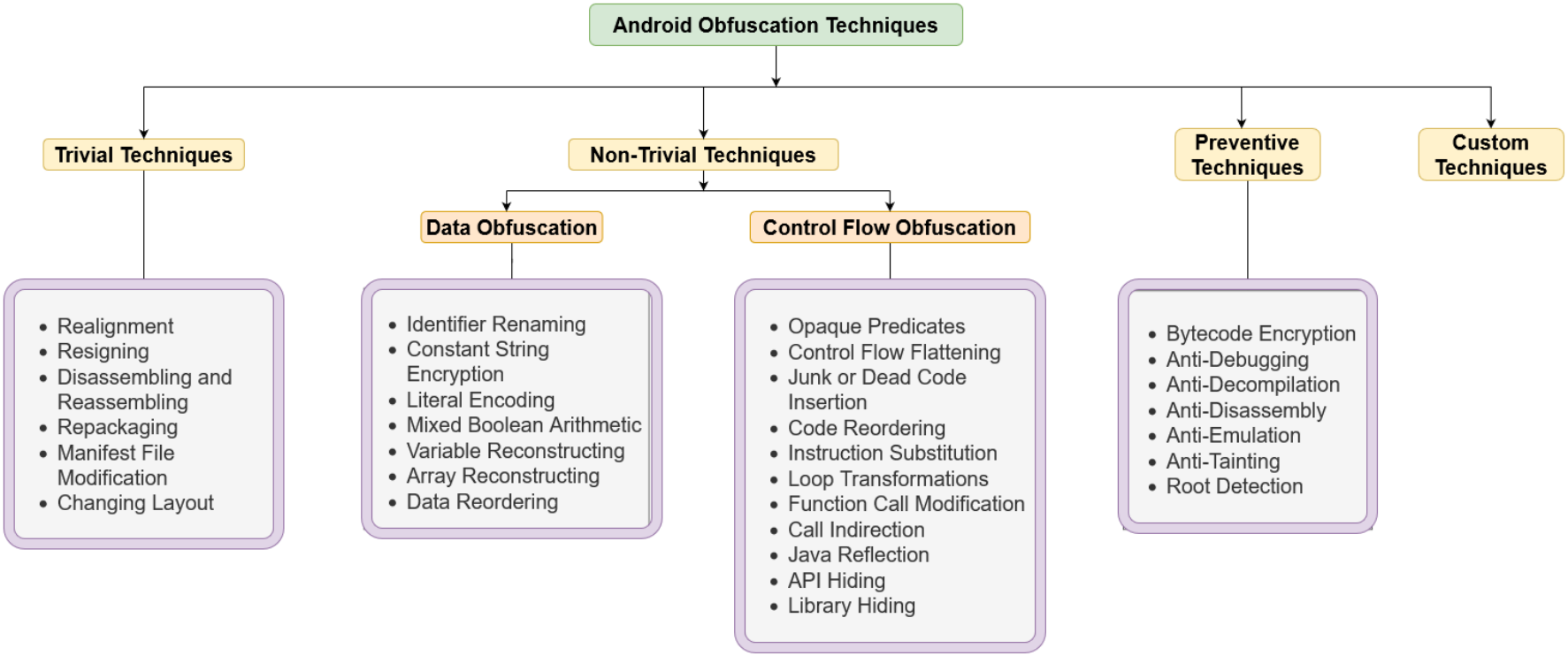


Figure 2.3: Android Obfuscation Techniques

when we hash the realigned apk, it will produce a different output. That way, malware can evade some detection engines that use signatures relying on the hash outputs of the whole application.

- **Resigning**: Android applications must be signed digitally with the developer's certificate as the last step while building APKs to be installed on devices. If a detection engine creates signatures based on the certificates of the apks, resigning the apks may evade detection. This technique is trivial and ineffective, yet it can be considered an obfuscation method.

- **Disassembling and Reassembling**: In this technique, classes.dex file is disassembled and reassembled by using various tools. This process creates a different but functionally equivalent version of the application due to the artifacts left by the disassembling itself. Hence, it gives a different digest [51]. Even though Disassembling and Reassembling is a trivial technique, it can fool detection engines that use signatures based on the classes.dex file of the applications.

- **Repackaging**: Repackaging is a widely-used obfuscation method employed by malware developers on the Android platform. Past research on repackaging shows that 86% of Android malware is contained in repackaged applications [52]. In [53], researchers reported that 30% of the Google Play Store applications are either plagiarized, repacked, or cloned. Likely in [54], it is stated that 77% of the top 50 free applications in the Google Play Store are being repackaged with a malicious payload and distributed on other alternative application markets.

  In this technique, particularly popular applications are targeted and reversed. During the reverse engineering process, apk is disassembled first using tools such as apktool and baksmali, and the smali code of the application is obtained. Then malware authors modify the source code, such as injecting junk code or malicious payload and then recompile and repack the application into a functional equivalent one to the original apk [55]. Finally, the resulting apk gets distributed over third-party markets, which mostly have fewer security measures. By doing so, repacked applications may evade detection engines and security protections of third-party markets and use the reputation of popular applications to trick users into downloading apks with malicious payloads. Malicious payloads can be used for various purposes, including but not limited to social engineering, fraud, and spying on users' activities such as accessing their messages, call history, contacts, and device location. Apart from injecting malicious payload, malware authors also use repackaging for economic gain, especially to receive purchase and advertisement revenue of popular applications.

- **Manifest File Modification**: In Section 2.1.1, we explained the role of AndroidManifest.xml file, which specifies the permissions, components, and features of the Android applications. Since the manifest file corresponds to the configuration of an application, it is also used by detection engines to create signatures and detect malicious activities. While compiling an APK, the manifest file gets compiled into a binary XML file. Manifest File Modification leverages this transformation to obfuscate and optimize binary XML file by renaming the file's activities and services to non-readable text to avoid detection [56].

- **Changing Layout**: This obfuscation technique aims to avoid detection by transforming the application's layout. The transformations include modifying the source and binary structure of the APK. Changing the output format, removing comments, and debug information can be classified under Changing Layout obfuscation method. However, this method is ineffective against modern malware detection engines and can be considered outdated.

2.2.1.2 Non-Trivial Techniques

Non-trivial obfuscation techniques include a variety of methods that are more effective against reverse engineering than trivial techniques. These techniques change the source code of the applications while preserving their original functionality. Non-trivial methods are generally complex and require at least a moderate level of code development knowledge. Non-trivial obfuscation methods can be grouped under two categories as data and control flow obfuscations.

Data Obfuscation

Data obfuscation methods target the data structures in the code. These methods are used to complicate the usage purpose of data fields to make the code harder to understand. Data obfuscation methods can be specified as follows:

- **Identifier Renaming**: In programming, it is standard practice to give names that reflect code semantics to identifiers to improve understandability and readability. This practice allows a group of developers to quickly collaborate on the same code to find and correct flaws or add new features later. On the other hand, meaningful identifier names allow reverse engineers to quickly comprehend the code logic and discover the target functions, which is not desired in the case of malware development. Therefore, the identifier renaming technique changes the identifiers, including classes, methods, fields, variables, and packages with meaningless names to perplex the analyst.

  Although various renaming strategies can be used for this technique, the most common one is to choose from random short strings which do not have any semantic value in lexicographic order, such as 'a', 'b', 'aa', 'ac'. Choosing from short strings during renaming also optimizes the code, shortening the bytecode used for identifiers. Moreover, some renaming strategies can be used to mislead the reverse engineer by choosing identifiers that reflect incorrect semantic information about the code. In short, identifier renaming is a popular and effective obfuscation technique that modifies the whole naming format of the code [57].

- **Constant String Encryption**: Constant strings in the source code of an application may contain sensitive and private information, e.g., passwords, cryptographic keys, or server addresses. An analyst can also locate target functions by analyzing the context in which strings are used. Thus, developers employ constant string encryption as a protection against reverse engineering so that strings are decrypted just prior to their use. For encryption, different algorithms are used. The common and basic way of string encryption is to use XOR encryption. Nonetheless, cryptographic algorithms such as DES and AES can

also be applied for string encryption. We should point out that the purpose of string encryption is not to follow the encryption standards; instead, it is to obscure the program against reversing. This technique is effective against static analysis. However, it can be defeated by dynamic analysis [58].

- **Literal Encoding**: This technique is similar to constant string encryption. It is used to hide constant variables. However, literal encoding utilizes basic functions that generate the constant during runtime instead of encryption. For instance, a string object can be generated during runtime; thus, an analyst will not be able to retrieve its value by looking at the binary. Constant values can also be encoded using opaque expressions that always have a fixed value during program execution [50].

- **Mixed Boolean Arithmetic**: This obfuscation technique is presented in [59] and utilizes arithmetic and Boolean operations to encode data. For instance, a linear equation X = a + b can be encoded as $X = (a \oplus b) + 2(a^b)$ by using mixed boolean arithmetic obfuscation method [50].

- **Variable Reconstructing**: This technique suggests reconstructing variables to make it confusing for the analyst to identify variables during reverse engineering. Variable reconstructing can be achieved by splitting or merging variables. Variable splitting splits a variable that will be reassembled from its divided pieces during execution into two or more variables. Data types such as integer, string, and Boolean variables can be obfuscated with this technique [42]. Contrary to variable splitting, variable merging combines multiple variables into a single variable. For instance, n scalar variable, $v_1, v_2, ..., v_n$ can be merged into a single array V_n or even promoted as objects. We should point out that only variables with the same type can be merged or split with this technique.

- **Array Reconstructing**: Array reconstructing technique is very similar to variable reconstructing. The difference is that this technique targets the data stored in arrays. Like variable reconstructing, arrays can be divided into two or more subarrays, and multiple arrays can be combined into one array. Moreover, array folding (i.e., increasing the array dimensions) and array flattening (i.e., decreasing the array dimensions) can also be used to obfuscate arrays [60].

- **Data Reordering**: Data locality states that logically related items are located physically close to each other in the binary and provides an understanding of the code during analysis. This obfuscation technique aims to decrease locality and make the analysis harder by randomizing the order of data declarations inside a program. This technique includes (a) Method reordering, which randomizes method declarations in the code, (b) Instance variable reordering, which randomizes the declarations of instance variables inside a class, and (c) Array reordering, which uses a mapping function to randomize data inside arrays [60].

Control Flow Obfuscation

Control flow obfuscation methods target the logic and control flow of Android applications through various modifications to perplex the analyst. Basically, these methods break up related functional blocks and combine unrelated functional blocks to thwart

static analysis and make reverse engineering more difficult. Control flow obfuscation methods can be described as follows:

- **Opaque Predicates**: A predicate is a Boolean-valued function that is called opaque if the result is known by the programmer in advance but cannot be resolved by an analyst or a deobfuscator statically [61]. Opaque predicates add complexity to the source code of an application and make the reversing and analysis harder or even impossible in some cases without executing the application. They are used with conditional jumps depending on the result of opaque predicates. Therefore, this obfuscation method ensures that a branch always gets executed while making the static analysis efforts obsolete. Customized versions of this obfuscation method can be found in the literature [62, 63].
- **Control Flow Flattening**: This technique aims to hide the relationships between basic blocks in source code so that the control flow cannot be determined by static analysis. Control flow flattening divides basic blocks in a source code and puts them in an endless loop with a switch statement. This switch statement is also called the dispatcher and regulates the application's control flow through the dispatch variable during execution. By doing so, the flow of application becomes significantly harder to follow for the analyst, and reversing requires more time and effort [60].
- **Junk or Dead Code Insertion**: The dead code refers to code blocks that are never executed because they are not or cannot be accessed in the control flow graph [39]. They are mainly used with opaque predicates to prevent their execution. On the other hand, junk code refers to code sequences that are executed but do not affect the program's state and functioning. Adding nop instruction is a well-known and widespread junk code insertion technique that does not change the application functionality. Inserting junk or dead code increases the amount of the source code and adds complexity for the analyst. This obfuscation method is mainly used to evade detection engines based on instruction (or opcode) sequences analysis; however, it is considered a weak technique by itself [50].
- **Code Reordering**: The goal of code reordering is to change the sequence of instructions in the smali methods of an application. In order to retain the original functionality and execution order of the application, the instructions are reordered utilizing goto instructions. This method is applied to methods that do not contain any control jumps. Code reordering is a popular and effective technique employed by malware authors to evade detection engines that are based on signatures or instruction sequence analysis [64].
- **Instruction Substitution**: The functionality of an application can be implemented in a variety of ways. Also, the instruction set of most architectures provides instructions that are syntactically different but semantically equivalent [50]. Therefore, instructions of an application can be substituted with functionally equivalent ones to evade detection engines based on instruction (or opcode) sequence analysis and signatures.
- **Loop Transformations**: Loop transformations were initially designed to optimize the performance and memory usage of loops. However, some loop trans-

formations introduce complexity to the structure of the code and can be used as an obfuscation method. Most popular loop transformations include loop tiling, loop unrolling, and loop fission. Loop tiling breaks up a loop into inner loops to optimize the loop cache. Loop unrolling replicates the code inside the loop one or more times to decrease the number of loop iterations. Loop fission branches a loop into multiple loops with the same iteration space [42].

- **Function Call Modification**: This obfuscation technique includes the removal or addition of function calls. The removal of a function call refers to the process of replacing all calls to a function with the function's body and then removing the function. On the other hand, the addition of function calls refers to the process of creating a function from instruction sequences and then replacing these sequences with the call to the functions. Removal and addition of function calls are also known as code inlining and outlining in the literature [60]. These are typical optimization techniques employed by the compiler. Nonetheless, function call modifications introduce complexity to source code and make the code more difficult for analysts to understand.

- **Call Indirection**: Some detection engines identify malware based on the signatures created from the call graphs of the application. Call indirection methods are primarily used to evade these malware detection engines through a simple manipulation of the call graph of an application. In this technique, a method call in the code gets redirected to a proxy method that calls the same original method [65].

- **Java Reflection**: Reflection is a popular feature of the Java programming language that allows developers to interact with the application in various ways by creating, modifying, and accessing an object at runtime and dynamically invoking methods. However, it can also be used as an obfuscation method because of the reflective invocations, particularly those calling other functions. This technique is mainly used to evade static analysis since reflective invocations implicitly transfer execution flow to the related code segment. Reflection can also be used to hide and obfuscate sensitive library and API calls [50].

- **API Hiding**: Applications can be represented as sequences of Application Programming Interface (API) methods that provide interaction between the application and the operating system (Android environment in our case). API sequences and patterns can reveal a lot of information about the application. Therefore, various detection engines and studies focus on API call sequences to identify malware. This technique is used to hide the usage patterns of API calls to evade such detection engines [50].

- **Library Hiding**: Android applications widely utilize the system and third-party libraries since they provide various features and functionalities to developers. Library calls in the code present helpful information to an analyst and can also be used to identify malware. This technique includes various schemes, such as recreating custom versions of the library to hide library calls and prevent library information leakage [42].

2.2.1.3 Preventive Techniques

Preventive techniques aim to prevent static or dynamic analysis of an application or at least make the analysis challenging. The primary goal of these techniques is to detect the analysis environment through the environment and emulator-related properties such as static properties and runtime sensor information [50]. After the detection, preventive techniques employ various methods to disrupt or halt the analysis of an application. These techniques are out of the scope of our study; however, popular preventive techniques can be briefly described as follows:

- **Bytecode Encryption**: Encryption is used to secure the application against static analysis. The encrypted code gets decrypted during runtime. It is used mainly by malware authors to hide their malicious payload [65].
- **Anti-Debugging**: This technique is used to protect the application against debugging by tricking debuggers to make the analysis more difficult.
- **Anti-Decompilation**: This method prevents Dalvik or Java bytecode from being reverse engineered into high-level programming constructs to make the analysis harder.
- **Anti-Disassembly**: These approaches use special code included in a file to trick disassembly tools into providing inaccurate outputs for the application code to confuse the analysts.
- **Anti-Emulation**: Malware authors mainly use this technique to detect the emulation or virtual environments so that when the application gets executed in such a controlled environment, it can hide its malicious behavior to thwart analysis.
- **Anti-Tainting**: This technique is used to protect the application against taint analysis to evade information flow leakage.
- **Root Detection**: This method is used to detect if the application is running on a rooted device.

2.2.1.4 Custom Techniques

As we can see, numerous different obfuscation techniques exist for the Android environment. The variance and abundance of these techniques allow a developer to mix, customize, and implement their own customized obfuscation. Also, we should point out that even though we explained obfuscation methods separately, one can implement several obfuscation methods to the same application. Besides common trivial, non-trivial, and preventive techniques, custom obfuscation techniques have been used in the wild. Custom obfuscation techniques include methods such as path obfuscation/phishing, hiding packages in raw resources or assets, and loading non-Dalvik code [66].

Given the complexity of the domain, compiling a dataset that includes practically all of the known obfuscation strategies is impractical. Therefore, we limited the scope of the present study and focused on detecting the popular nine non-trivial obfuscation

methods. We then applied these obfuscation methods to our collected apks to build our dataset. The utilized obfuscation methods for this study and their implementation will be detailed in section 3.3.

2.2.2 Android Obfuscation Tools

In the previous section, we explained various obfuscation techniques and how easy it is to mix and customize these techniques. The abundance of obfuscation techniques and their practicality in implementation are also reflected in the number of obfuscation tools. Hence, there are various commercial and open-source obfuscation tools available for use. Since it is infeasible to mention all the obfuscation tools in the wild, we will explain the most used commercial and open-source obfuscation tools and their provided options from the literature in this section. Popular commercial obfuscation tools can be stated as follows:

- **Allatori**: Allatori [67] is a commercial obfuscator from Smardec company. As most obfuscators do, Allatori provides renaming of identifiers, methods, classes, and packages. The tool gives the opportunity to customize the new names during renaming. This feature allows developers to use their own unique renaming policies. Moreover, Allatori lets developers modify the program code with methods such as code removal, junk code insertion, and control-flow modifications. The tool can also modify the package hierarchy with repackaging and package flattening techniques [19]. Additionally, Allatori provides constant string encryption method, and strings are obfuscated and decoded at runtime.
- **DashO**: DashO [68] is a commercial obfuscator developed by PreEmptive Solutions. DashO provides obfuscation methods similar to Allatori. The tool allows the renaming of identifiers, including classes, fields, methods, and interfaces. Customizable renaming schemes are also possible in DashO. The obfuscator implements control flow modifications to add confusion to the source code and enables constant string encryption for developers. Furthermore, DashO offers two optimization options removal (pruning) and bytecode optimization. Pruning is used to remove the unused classes in an application. On the other hand, bytecode optimization is used to optimize the code to make it faster and smaller [69].
- **DexProtector**: DexProtector [70] is another commercial obfuscation tool used against reverse engineering for the Android environment. DexProtector works directly on Dalvik bytecode. The obfuscator provides encryption for constant strings and allows encryption of classes, assets, and native code. DexProtector also offers identifier renaming. Moreover, the tool can be utilized to hide the APIs called within an Android application [71]. Additionally, it provides environment checks and device attestation options. However, DexProtector does not provide control-flow modifications.
- **DexGuard**: DexGuard [72] is a commercialized version of ProGuard obfuscator, which will be detailed below, with extended features. The extended features of DexGuard include string and class encryption and the possibility to use non-ASCII characters while renaming identifiers. The obfuscator works on

Java code instead of bytecode. Similar to DexProtector, DexGuard offers identifier renaming and API hiding techniques but does not provide control-flow modifications. Samples that are obfuscated with DexGuard are reported to be challenging to reverse engineer [73].

Most popular open-source obfuscators are as follows:

- **ProGuard**: ProGuard [74] is the most well-known Android obfuscation tool since it is the default obfuscator provided with the Android SDK package. Even though it is essentially an obfuscator, ProGuard also offers optimizing, shrinking, and Java class file pre-verification options. The tool provides identifier renaming method for methods, classes, fields, and packages. ProGuard uses the alphabet by default during renaming, but it is also possible to supply custom renaming schemes. However, ProGuard does not support constant string encryption and control flow modification techniques. Although it is a widely used tool, the obfuscation methods provided by ProGuard are limited and do not introduce heavy changes into the source code. Hence, the obfuscator cannot provide a sufficient level of overall security to stop reverse engineering but presents another obstacle to pass for attackers [57].
- **ADAM**: ADAM [75] is an open-source obfuscator developed to generate different malware variants from original malware samples while retaining the exact malicious behavior. The obfuscation methods supported by ADAM contain both trivial and non-trivial techniques. Provided trivial obfuscation methods such as realignment, resigning, disassembling and reassembling, and repackaging works on apk files without modifying the source code. ADAM also offers non-trivial obfuscation techniques that modify the source code, including identifier renaming, control-flow modifications, defunct code insertion, and constant string encryption.
- **AAMO**: AAMO [76] is an open-source obfuscation tool with various obfuscation options. The tool provides obfuscation techniques under five categories as follows: Android-specific, simple control-flow modifications, advanced control-flow modifications, renaming, and encryption. Android-specific methods include trivial techniques such as repackaging, realignment, and reassembly. The tool offers different control-flow modifications such as junk code insertion and call indirection under simple and advanced control-flow modifications categories. Unlike commercial tools, AAMO allows developers to apply control-flow modifications separately. Moreover, AAMO offers renaming of non-code files, resources, fields, methods, and packages and encryption of asset files, native code, and constant strings. The downside of this obfuscator is that it is written in Python 2, and Python 2 is no longer supported; hence the tool needs to get updated to work properly.
- **Obfuscapk**: Obfuscapk [77] is the latest developed open-source obfuscation tool that we came across in the literature. It is written in Python 3 and also utilizes apktool to decompile apk files for obfuscating applications without needing the source code. The tool covers all the obfuscation modifications offered by the previously mentioned commercial and open-source obfuscators. Additionally, it supports advanced obfuscation techniques such as encryption of

native libraries and advanced reflection. The obfuscation methods provided by obfuscapk are categorized as Trivial, Rename, Encryption, Code, Resources, and Other. Trivial techniques supported by the tool are similar to other tools. Rename category includes renaming of classes, fields, and methods. Encryption options of the tool cover encryption of constant strings and resources such as assets and libraries. Obfuscapk also offers various obfuscation methods that affect the code and change control flow. Similar to AAMO, obfuscapk also allows developers to implement control-flow modifications separately.

We mentioned the most popular commercial and open-source obfuscation tools. There are many other obfuscators available, and each tool covers different obfuscation methods with different configurations. Commercial obfuscators were not suitable candidates for our study since they are proprietary and do not offer evaluation versions in most cases. Moreover, they do not separate control-flow modifications; instead, they offer those modifications as a single option. Since we also want to investigate the performance of LSTM on different control-flow modifications, we focused on open-source obfuscators, which provide control-flow obfuscations separately. Both ADAM and AAMO were not updated since the date they were released. On the other hand, ADAM does not support advanced obfuscation techniques. We also mentioned that AAMO does not work properly since it is written in Python 2. Since obfuscapk is already developed to overcome these limitations of previous obfuscation tools [77], we used obfuscapk in the present study. It covers all the obfuscation methods provided by other tools, such as identifier renaming, code reordering, string encryption, and junk code insertion. Also, it supports separate control-flow modifications. Moreover, the tool can be used from the command line, which provides practicality for obfuscating apks and building datasets. The implementation details of obfuscapk and the obfuscation methods we employed in this thesis will be detailed in the Datasets section under Methodology.

2.3 Signature-based Methodologies

We discussed that application developers use obfuscation methods to protect their intellectual property and optimize their code, whereas malware authors use obfuscation for evading detection and digital forensics. As a result, various studies focus on the different aspects of obfuscation in the literature, including clone detection, repackaging detection, malware detection, and third-party library detection. The mentioned detection research follows a similar historical development, whether for detecting malware or clone of a program. The detection methods started with signature-based techniques. In the signature-based approach, signatures are created from specific patterns and strings. Then the input and available signatures get compared for detection. In [20], the authors presented a malware detection method that employs signatures. Researchers divided classes of apks into modules by using Class Dependence Graph as a first step to extract malware signatures. Then, they analyze the combination of modules and their API calls and determine the combinations with the highest inter-module similarity to locate malware modules. Lastly, malware family signatures are created based on the selected API call combinations that are located and marked as malware in the previous step. Their signature-based model reached 94% of accuracy

for detecting malware. In a similar study [21], researchers proposed a signature-based system called DroidAnalytics to detect and analyze malware. The system has various features and automatically collects and presents information about the analyzed application on the opcode level. The system has a Signature Generator module which employs three-level signal generation. Three levels of signature generation correspond to the application, class/dynamic payload, and method signatures, respectively. DroidAnalytics perform malware and repackaging detection based on the created signatures. In a different study [78], authors suggested a statistical signature method called AndroSimilar for detecting unknown versions of existing malware, mainly generated through code obfuscation methods (e.g., repackaging). Authors generated signatures from statistically improbable features such as normalized entropy and built a signature database for malicious applications. Then the signatures of the investigated applications are compared with the signatures in the database to detect malware variants based on the similarity percentage. The authors also employed several code obfuscation techniques to change the code signatures and test the proposed model. The model achieved a 60% detection rate for obfuscated samples.

In [79], Chen et al. introduce a method that employs signatures and NiCad, a near-miss clone detector, to detect Android malware. To create signatures, researchers choose a subset of malicious applications as a training set, and then they identify the clone classes in training set by using NiCad. NiCad utilizes an optimized longest common subsequence algorithm and compares the source code line by line to detect clones. Then the clone classes are used as signatures to find similar malware in the rest of the malicious applications. Another study [80] also uses signatures and clone detection for identifying Android malware variants. Researchers used a language called MAIL to detect clones, a new Malware Analysis and Intermediate Language. MAIL provides an abstract representation for native Android binaries that can be used to locate code clones, which helps detect malware. Hence, before creating the signatures, researchers converted Android applications into MAIL. Then, they generated the Control Flow Graphs (CFG) of the MAIL programs and used MAIL patterns to create signatures. In the signature generation process, they build signatures for each function and block in MAIL programs. Therefore, each signature includes a function signature and a collection of block signatures. Using an optimized version of the largest common substring comparison with a threshold, researchers compared the signatures of Android applications with the generated signatures to detect malware and achieved a detection rate of 97.85%. In [81], the authors focus on detecting third-party libraries using signatures. They proposed a system called LibID to identify third-party libraries from Android binaries accurately. LibID has two detection methods: LibID-S and LibID-A. In LibID-S, basic blocks are extracted from application binaries, and then signatures are generated for extracted basic blocks. Later on, researchers include some class features to block signatures and generate class signatures. Class signatures are used for class and dependency matching. On the other hand, in LibD-A, some additional information such as method calls, class inheritance, and class interfaces are recorded. Then dependency graphs are built based on the recorded information, and these graphs are used in the matching phase to provide more accurate library detection. The authors tested their system against several obfuscation techniques; however, they stated that their system is not robust against advanced obfuscation techniques.

The signature-based detection methods achieved high detection rates in some cases; nevertheless, they still have limitations that make these methods obsolete. Obfuscation is a significant problem in detection methods based on signatures. As we explained in Section 2.2.1, numerous different obfuscation techniques can be employed to change the source code of an Android application. Also, applying obfuscation takes little time and effort for malicious actors to evade detection. On the other hand, generating signatures is a very time-consuming and laborious process. Therefore, it is infeasible and almost impossible to generate signatures for every existing obfuscation method. Some signature-based detection research, such as [78] and [81], tried to build robust systems against obfuscation. However, the resilience of those systems is specific to chosen obfuscation tools and their configuration. Hence, even though they achieve high detection rates for their datasets, they are not generalizable considering the vast number of obfuscation methods. Also, signatures can only detect instances of known samples. Signature-based methods make the system vulnerable to new malware samples in malware detection cases. Another problem of signature-based detection systems is scalability. We mentioned that both the Android applications and malware are huge in numbers. Therefore, creating a signature database that covers sufficient benign and malware samples will require significantly large storage spaces, which is also impractical. Because of the limitations mentioned above, the research community focused on new, more efficient detection methods and the detection paradigm shifted towards Machine Learning (ML) based detection methods which will be detailed in the next section.

2.4 Machine Learning Methodologies

The previous section mentioned the limitations of signature-based detection methodologies against building automated, reliable, and generalizable detection systems. The increase in the number of Android applications, new malware types targeting Android, and the new obfuscation techniques make signature-based methods out-of-date since the manual work needed to create signatures and cover all these new samples is nearly impossible. Hence, the research community turned its focus to Machine Learning (ML) methodologies first and then the deep learning methodologies. In this section, we will explain ML methodologies. ML is a branch of artificial intelligence that allows researchers to develop systems that mimic human learning by using data and algorithms rather than explicit programming. ML methodologies consist of various techniques that make predictions based on past observations. The typical lifecycle for building an ML system includes several sequential steps: data collection, data preprocessing, feature extraction, model training, model evaluation, and model deployment [82].

Scientists and researchers from several areas focusing on malware, clone, and third-party library detection utilized various ML algorithms to build reliable and accurate detection systems. According to the survey [83], Logistic regression, Naïve Bayes, Support Vector Machine, k-Nearest Neighbor, Decision Tree, and Random forest ML algorithms are suitable and widely used algorithms in the literature for detecting Android malware. Logistic regression is a classification algorithm that is used to predict a binary outcome. This binary outcome is based on the relationship between

the dependent variable and a set of independent variables. Naïve Bayes is another classification algorithm that assumes all features are independent and can be used to express and analyze uncertain or probabilistic events [83]. Support Vector Machine (SVM) is an ML algorithm that can be used for both regression and classification. The SVM aims to find a hyperplane in an N-dimensional space that classifies the data points precisely where N is the number of features. SVM is also one of the common kernel learning methods used for nonlinear classification. k-Nearest Neighbor (kNN) is a supervised ML algorithm used for classification. kNN classifies a new instance using a similarity distance (e.g., Euclidean) in comparison to the instances used in the training phase and the votes provided by its K nearest neighbor during the testing phase [84]. The Decision Tree is a supervised learning algorithm that can perform regression and classification by utilizing tree structure. The Decision Tree learns basic decision rules using labeled training data to predict the class or value of a target variable. Another supervised ML algorithm called Random forest combines multiple decision trees for better accuracy. The algorithm makes predictions based on the majority of the decision trees so that even if some trees produce false predictions, the majority will increase the overall accuracy.

In [85], the authors employed ML methodologies for Android malware detection. They focused on and extracted several features from the AndroidManifest.xml file. The feature set of the study consists of permissions required for the application and the features under the uses-features group. Then authors experimented with different ML algorithms, including Naïve Bayes, SVM, kNN, J48, and Random forest with different configurations and parameters. According to their results, Random Forest with 100 trees achieved the best detection accuracy of 94.83% among the tested algorithms. Also, SVM with normalized polynomial kernel achieved a detection rate of 94.50%. In a similar study [86], researchers proposed a system called DREBIN that utilizes SVM. They extracted as many features as possible by combining static analysis of the Android manifest file and disassembled code for better performance. The extracted features include hardware components, requested and used permissions, app components, filtered intents, restricted and suspicious API calls, and network addresses. Then researchers embedded these features into a vector space and fed vectors into the SVM model for malware detection. The proposed system achieved a detection accuracy of 94%. In another study [87], the authors propose two ML-based methods for detecting malware on the Android platform. In the first approach, permission names are used as features. On the other hand, features such as API and system calls, services, and methods are extracted from the source code in the second approach. Then authors use the bag-of-words model to generate feature vectors to train the models. The authors experimented with different ML algorithms in both approaches, including C4.5decision tree, random forest, SVM with SMO, and logistic regression. Apart from the previously mentioned studies, popular static features used by ML-based Android malware detection studies are permissions, API calls, opcode sequences, function call graphs, and their combinations [88].

Similar trends can also be seen in the studies focusing on third-party library detection and clone detection in Android. In [89], the authors proposed a system called AdDetect, which utilizes semantic analysis and machine learning to detect ad libraries. The authors extracted semantic features, including usage of Android components, permissions, and APIs, from each module and represented those features as vectors. Then,

the authors detect ad libraries by training an SVM model with the generated feature vectors. Another similar study [90] presents a methodology called PEDAL which also focuses on ad library detection problems by utilizing the SVM algorithm. These two studies are the pioneers of employing ML methodologies in the third-party library detection field. However, both studies only focus on detecting advertising libraries. Likewise, applications of ML methodologies, including Logistic regression, K-means clustering, Random Forest, Rotation Forest, SVM, and Naïve Bayes have been utilized by researchers who focus on detecting clones of Android applications [91].

ML methodologies performed better than signature-based methods; however, they also have some limitations for building automated and reliable detection systems. The major problem in ML methodologies is the feature extraction process. First of all, feature extraction requires human intervention and the knowledge and expertise of the analyst. Moreover, some studies prefer to extract as many features as possible to improve the accuracy of the detection systems, which needs intense effort. Another issue to consider is scalability. If the datasets are large, some of the feature extraction techniques will not be able to run. On the other hand, even though supervised ML-based detection systems achieve higher accuracy rates, the detection systems are not robust against new samples and variants due to the nature of supervised learning. Hence, more recently research community looked for alternatives to overcome the limitations of ML methodologies and turned their focus to deep learning and neural networks. Since deep learning methodologies can be used without feature extraction, they are good candidates for building automated and reliable detection systems. Also, neural networks perform better than ML methodologies in some tasks. We will continue our discussion by explaining neural networks and deep learning-based detection methodologies in the following section.

2.5 Deep Learning Methodologies

In this section, we will present the deep learning-based, specifically LSTM-based, detection studies that address the limitations of the previous ML-based detection systems. Nevertheless, before presenting the studies, we will provide some background information on Neural Networks and Natural Language Processing (NLP).

2.5.1 Neural Networks

Artificial Neural Networks (ANNs), or neural networks, are a subset of machine learning that can learn from complex or imprecise data. ANNs are inspired by the structure and functionality of biological neural networks and consist of linked nodes. Hence, a node in the ANN architecture corresponds to a neuron in a biological neural network. Compared to a biological neuron, a node works in a much simpler way. However, nodes can learn, generalize the training data, derive results from complex data, and make predictions. The learning process of ANN can be summarized as follows: In a neural network, each node has a numerical value called weight that is linked to an input for making predictions. Depending on whether or not a prediction is correct, the networks update the weights of links between nodes. A mechanism

called backpropagation is commonly used to update the weights in neural networks. For a single input-output example, backpropagation calculates the gradient of the loss function with respect to the neural network's weights. Then, the calculation, which is the loss value, proceeds backward through the network nodes. Consequently, the weights of the nodes get updated [92].

There are various types of ANN architectures, including but not limited to Convolutional Neural Networks (CNN), Modular Neural Networks (MNN), Recurrent Neural Networks (RNN), Generative Adversarial Networks (GAN), Deep Neural Networks (DNN), Spiking Neural Networks (SNN), Physical Neural Networks (PNN), and Hybrids [93]. Each ANN type is specialized and performs better for a particular learning task. For example, DNNs have larger memory capacity and can break down big problems into a series of smaller problems for a more efficient learning process. CNNs are suitable for processing visual data and two-dimensional data in general. They achieve high-performance results in both image and pattern recognition tasks. On the other hand, RNNs effectively deal with sequential data and perform well in text recognition, speech recognition, and language modeling. In the scope of our study, we are interested in RNNs specifically. Thus, we will continue our discussion by explaining RNN architectures.

Recurrent Neural Networks (RNN)

Recurrent Neural Network (RNN) is a special type of artificial neural network. In classic neural networks, inputs and outputs are assumed to be independent of one another. However, the output of an RNN depends not only on the current input but also on the prior inputs. This dependence is due to the architecture of RNN, which allows a node in one layer to connect nodes in previous layers. This connection provides a hidden state of the previous phase to be used with the input of the current phase to calculate the output of the current phase. The involvement of previous inputs and hidden states provides memory to RNN architectures which is the main difference between RNN and other neural networks. The presence of memory makes RNN suitable for working with sequential data, and they are primarily used in natural language processing (NLP), language translation, image captioning, and speech recognition.

RNNs are further distinguished by the fact that their parameters are shared across all layers of the network. Moreover, RNNs use the BPTT (Backpropagation Through Time) algorithm to determine the gradients. This is slightly different from traditional backpropagation because it is unique to the sequence data. The underlying principle of BPTT is the same as traditional backpropagation, where the model trains itself by computing errors and updating the weights of nodes. The difference is that the BPTT sums errors each time [94]. This difference creates two problems for RNNs: exploding gradients and vanishing gradients. Exploding gradients arise when the gradient is too large, creating an unstable model by assigning high importance to the weights abruptly. Luckily, truncating or squashing the gradients is a simple solution to this problem [1]. On the other hand, when the values of a gradient are too small, vanishing gradients occur. When the gradient is too small, it continues to become smaller and update the weight parameters until they become negligible, and eventually, nodes and the model stop learning. The vanishing gradient problem causes RNN to have short-term memory, making them less effective on long sequential data and decreasing their performance. Even though this problem is not as easy to solve as exploding gradi-

ents, specialized RNN architectures such as Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) were developed to overcome the problems of Recurrent Neural Networks.

Although both LSTM and GRU are developed to address the vanishing gradient problem, there are some differences between the two. The differences concerning our study are as follows: The GRU lacks internal memory and an output gate, both of which are present in LSTM. Moreover, LSTM is more accurate on larger datasets and is a better candidate while dealing with large sequences to achieve high accuracy values. Also, LSTM has been successfully employed in various detection studies, which will be described in section 2.5.3. Therefore, we utilized LSTM in our study and will continue our discussion by detailing the LSTM architecture.

Long-Short Term Memory (LSTM)

Long Short-Term Memory (LSTM) networks are a modified version of RNNs that are introduced to deal with the vanishing gradient problem. In the classic RNNs, if the context window is large or the previous phase affecting the current prediction is not recent, the RNNs cannot connect the information and make accurate predictions. However, LSTMs have an improved and modified architecture to overcome the vanishing gradient problem and decrease its effect on the model so that they have long short-term memory. Also, the output of a current step is dependent on three things which are cell state (the current long-term memory), hidden state (the output of the previous step), and the current input in the LSTM model, unlike RNNS [1]. LSTMs work well with long dependencies and can hold and remember information for a long time. This capability is due to their memory cells which are visualized in Figure 2.4 below. Therefore, they are suitable candidates for learning from long sequences of text and speech.

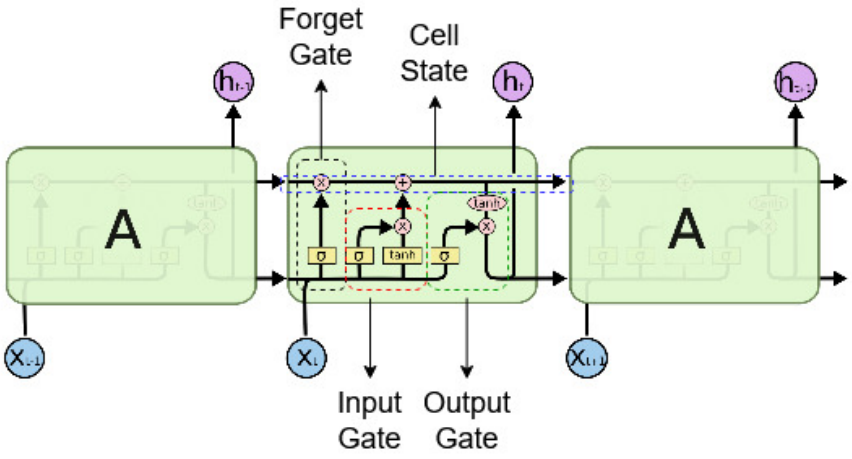


Figure 2.4: The architecture of an LSTM cell [1]

In LSTM architecture, there are internal mechanisms called memory cells. The cell state, which is a value that flows through cells to carry information, is a critical com-

ponent of the LSTM cell. LSTM cells can modify the cell state by adding or removing information through structures called gates. Gates control the information flow in the network and can also be thought of as filters that remove unimportant information. There are three different gates in an LSTM cell called forget, input, and output gate, as shown in Figure 2.4. The details of these gates are as follows:

- **Forget Gate**: Forget gate is the first segment of the LSTM cell. This gate basically decides which information to keep or discard. To make this decision, the previous hidden state (output of the previous cell) and the input of the current cell are fed into a sigmoid activation function to generate a vector that consists of elements between 0 and 1. In the context of forget gate, 0 means irrelevant information, whereas 1 means relevant information. Then the outputted values by the sigmoid function get pointwise multiplied by the previous cell state to create and update the current cell state. With the pointwise multiplication, parts of the cell state considered irrelevant by the model get multiplied by a number close to 0 so that they have a low effect on the succeeding steps. As a result, forget gate determines how much of the long-term memory information should be forgotten and how much should be kept [95].
- **Input Gate**: After the forget gate in an LSTM cell, the next segment is the input gate. This gate decides what new information must be added to the cell state. The input gate takes the input of the current cell and the previous hidden state as inputs, similar to forget gate. Then, using a tanh function, a vector is generated from the input of the current cell and the previous hidden state. This vector contains information about candidate values and specifies how much to update each element in the cell state. We should also mention that using the tanh function produces values between -1 and 1. The negative values in the context of the input gate are necessary and used to weaken the effect of a component in the cell state. There is also a sigmoid function that takes the same input as tanh, and it is used to specify which elements are worth remembering and should be added to the current cell state. As we explained in the forget gate, the sigmoid function generates a vector whose elements' values are in the interval of 0 and 1. In the input gate, values near 0 indicate the components of the cell state that should not be updated. After both the sigmoid and tanh functions produce their output vectors, these vectors get pointwise multiplied. This multiplication determines which values will be added to the current cell state [95]. Lastly, an addition operation updates the previous cell state with the new information from the input of the current cell and forms the current cell state. In short, the input gate determines what information is kept in long-term memory.
- **Output Gate**: The output gate is the last segment in an LSTM cell which acts as a filter to guarantee that the only necessary information is outputted. Output gate determines the output of the current cell (new hidden state) by using the current cell state, the output of the previous cell, and the input of the current cell. Like the input gate, the output gate also has a sigmoid and a tanh function. The tanh function is applied to the current cell state to push the values to be in the -1 and 1 interval. Then, the sigmoid function is used to create the filter vector by taking the input of the current cell and the output of the previous cell as inputs. The values of the filter vector lie between the 0 and 1 interval. Afterward, the filter vector and the output of the tanh function get pointwise

multiplied. This multiplication decides which components of the current cell state will be kept in the new hidden state [95].

The gates we explained above allow LSTMs to learn and grasp forward and backward dependencies in long sequences. We also mentioned that LSTMs are designed to overcome the vanishing gradient problem. The improvements in the architecture of LSTM and memory cells provide a long short-term memory, unlike RNNs. Therefore, LSTM is suitable for learning and identifying structural patterns and behavior in sequential data.

2.5.2 Natural Language Processing

Natural language processing (NLP) is a branch of computer science, artificial intelligence, and linguistics. The goal of NLP is to enable computers to understand and interpret text and spoken words in the same manner that humans can. NLP methods are employed in tasks including but not limited to speech recognition, part of speech tagging, word sense disambiguation, named entity recognition, co-reference resolution, sentiment analysis, language translation, and natural language generation [96].

The development of NLP started with linguistics. Firstly, classical linguistics took a rule-based approach to understanding natural languages. They formulated the syntax and semantics of natural languages as formal methods based on mathematical rules. However, early rule-based systems were not effective against increasing amounts of data due to the lack of computing power. Then with the advancements in computer science and computational tools, computational linguistics has emerged. Computational linguistics utilizes computers and software to understand better and model natural languages. The new computing power, large amounts of data, and the introduction of statistical and machine learning methods led the community to focus on data-driven statistical methods rather than the rule-based approach [97]. Involvement and acceptance of statistical methods in NLP created statistical natural language processing. These models can extract, label, and categorize components of text and speech data in a more automated manner than rule-based systems by utilizing ML algorithms such as Naïve Bayes, k-nearest neighbors, hidden Markov models, conditional random fields (CRFs), decision trees, random forests, and SVM [98]. However, with the advancement of neural models in recent years, the scientific community has embraced deep learning approaches, which show much promise for complex natural language processing tasks.

As we mentioned in the ML-based detection methodologies, traditional ML-based NLP approaches also suffer from a time-consuming, manual feature extraction process through a careful examination of a specific application. Nonetheless, DL-based supervised learning approaches provide more reliable and generalizable data representations since they are immensely data-driven. Moreover, DL methods also offer unsupervised learning, which is an essential NLP task due to the large volume of unlabeled data [99]. The DL-based approaches allow NLP systems to extract more accurate meaning from large amounts of unstructured, unlabeled text and speech data [96]. The previous section also explained the neural networks, especially LSTM, and their

advantages for handling long sequential data such as natural languages. Therefore, DL-based algorithms such as CNN, RNN, and LSTM have been used widely by the research community due to their better performance in NLP tasks.

We mentioned that NLP methods could be used in various tasks such as language modeling, semantic role labeling, text classification, machine translation, and speech recognition. In the scope of our study, we are more interested in language modeling and word embedding tasks of NLP. The technique of building a model to predict words or basic linguistic components based on prior words or components is known as language modeling [98]. Language modeling is crucial because it can grasp syntactic and semantic relationships between words or components. On the other hand, word embeddings are produced using language modeling techniques and consist of vectors with numeric components to represent words in lower-dimensional space. In the methods such as Bag of Words (BOW) and CountVectorizer, different words have different representations, resulting in large input vectors. On the other hand, word embeddings utilize word usage in context to represent semantically correlated words similarly [99]. Thus, word embeddings reduce the dimensionality of the input vectors. Moreover, it helps grasp semantics between words. Due to their advantages, word embeddings have become the standard input form of modern NLP systems.

In recent years, NLP techniques have also been employed to provide security in the Android environment. There are various valuable textual data such as definitions of applications, user scores, and reviews that can be found in Android application markets. Moreover, the source code of applications can also be considered textual data on which NLP techniques can be applied. Thus, NLP techniques provide mechanisms to comprehend and use such textual data to secure Android applications. In the survey [100], the authors investigate the usage of NLP methods in Android security. They state that NLP methods have been utilized in various security as follows: NLP can be used to identify the application behavior from textual descriptions to discover dangerous permissions in the application. NLP techniques can also be used to ensure users' privacy. Moreover, the authors present various studies that employ NLP methods for malware detection and clone detection in Android.

2.5.3 Detection Methodologies with Deep Learning and NLP

In the previous sections, we explained ML-based detection methods and their limitations in building robust, automated detection systems. We also described the neural network architectures and the advancements in NLP that can overcome the limitations of the previous detection systems. Therefore, the research community has applied deep learning methods and NLP techniques for detection systems, especially malware detection, in recent years.

Numerous studies in the literature have employed NLP DL models to build malware detection systems. Most popular DL algorithms that are applied in malware detection include Multilayer Perceptron (MLP), Deep Belief Networks (DBN), Autoencoders, CNN, and RNN [101]. The survey [101] states that malware detection studies mainly characterize applications using permissions, API calls, opcode sequences, control flow graphs, and Android bytecode. For example, in [102], researchers apply

NLP techniques and propose a framework that relies on API method call sequences to detect and classify malware in the Android environment. They built a CNN model for the study and achieved a detection accuracy of 96%-99%. In another study [103], Wu et used requested permissions extracted from the Android manifest file (AndroidManifest.xml) instead of API calls to detect malware. Then researchers used word2vec and represented the permissions as a one-hot vector to be fed into the LSTM model.

A further review of the NLP techniques reveals that opcode sequences are often used in malware detection. Deeprefiner [104] is a semantic-based, two-layered technique that addresses the problem of malware detection. The first layer of Deeprefiner conducts an initial detection by feeding vector representations of Android manifest files and all other XML files in the resources directory to an MLP model. Applications that the first detection layer cannot correctly classify are passed to the second detection layer for a more complete examination. In the second layer, the dex bytecode of the applications is represented as vectors by using Skip-Gram modeling. Then these vectors are forwarded to an LSTM model for classifying malware. In [105], the authors also utilized opcode sequences and LSTM for malware detection and family attribution. The authors also tested alternative neural network architectures and stated that LSTM achieved better results than other state-of-art models. In [106], researchers experimented with CNN and LSTM models. For the LSTM model, the authors used one-hot encoding to represent Android bytecode as vectors. Then generated vectors are inputted into the LSTM model. The suggested LSTM model achieves 70% accuracy for small datasets, whereas for large datasets, it achieves 95.3% accuracy.

As we can see from the literature, LSTM architecture is a suitable candidate for extracting meaningful information from long data sequences, even when this data is the source code of an Android application. Therefore, LSTM can be used to build automated and reliable detection systems that do not require manual human intervention at any stage.

2.6 Obfuscation Detection Techniques

The most relevant studies to our work in this thesis are the obfuscation detection techniques in Android. Therefore, this section will review the literature on detecting obfuscation in the Android environment. We explained previously that methods for detecting obfuscation have a similar historical development to other detection methodologies, especially malware detection. The detection methodologies started with signature-based methods. A signature is created from specific malware patterns and strings after a laborious process. Since it is easier to change the source code and evade the detection than to create a signature, signature-based methods became ineffective. Hence, the paradigm shifted towards ML-based detection methods. Even though ML-based detection methods performed better than signature-based strategies, the problem of feature engineering became an obstacle to reliable and generalizable detection. The research community embraced deep learning approaches, especially RNN and LSTM models, to overcome the limitations of previous methodologies, as we mentioned in the previous sections. Likewise, we can see similar methodology trends in obfuscation detection.

In [6], Wermke et al. proposed a signature-based obfuscation detection tool called OBFUSCAN. The authors employed several heuristics that are established directly from the source code of Proguard. Then based on these heuristics and particular configurations of Proguard, researchers generated signatures to detect obfuscation. They focus on detecting identifier renaming and method overloading obfuscation methods and debug info, annotations, and source file removal. In another study [7], researchers suggested a system that utilizes both signature-based and ML-based methodologies. Authors disassembled apk files and generated both signatures and features from disassembled Smali code. The proposed system used a signature-based approach to detect reflection and packing. On the other hand, the authors trained an SVM model with fixed-length feature vectors generated from N-grams to detect identifier renaming and string encryption. In a similar study [107], Wang et al. proposed a system to detect the applied obfuscation tools and their configuration using ML. The authors focused on detecting identifier renaming, string encryption, control flow obfuscation, and packing. They used ProGuard, Allatori, dashO, Legu, and Bangcle as obfuscators. They proposed a two-step detection method that utilized an SVM model. The used obfuscation tool is initially identified by leveraging the names of files, fields, classes, methods, and packages in the first step. Then, the second step determines the applied obfuscation method and the configuration of the detected obfuscator. In another study [108], the researchers present an obfuscation detection tool called AndroDet. The tool detects identifier renaming, string encryption, and control flow obfuscation. They used an online machine learning approach and achieved an accuracy of 92.02% for identifier renaming detection, 81.41% for string encryption detection, and 68.32% for control flow obfuscation detection. In another ML-based study [109], researchers suggested a framework for detecting class-level obfuscation. They focused on detecting identifier renaming, string encryption, control flow obfuscation, and reflection obfuscation methods. Researchers disassembled apk files and obtained Java class codes of applications. They also employed NLP techniques and used Paragraph Vector as embedding to generate vectors from the disassembled code. Then authors experimented with different ML models such as Random Forest, AdaBoost, Extra Trees, and Linear SVM.

As we can see, obfuscation detection methods in Android have also started with signature based-methods. Then, several studies employed ML techniques. However, these obfuscation detection systems mentioned previously have the same limitations as the other detection systems. Therefore, the LSTM model was also applied in several obfuscation detection studies. For example, in [110], the authors suggested a system for function-level obfuscation detection. They built models using Graph Convolutional Networks (GCN) and LSTM networks. Researchers focused on both x86 assembly code and Android applications. For Android, they examined detecting identifier renaming, string encryption, and control flow obfuscation. Even though they worked with LSTM, researchers extracted features and used supervised learning to train the models. Lastly, the most relevant study to ours is conducted by Bacci et al. [64]. They proposed two approaches feature-based and direct-classification-based. Authors disassembled applications and extracted opcode sequences to be used in both approaches. In the feature-based approach, authors experimented with Support Vector Machines (SVM), Random Forest (RF), and Multi Layer Perceptron (MLP) algorithms. On the other hand, they employed an LSTM model for direct classification. The work in [64] is the only research that employs both a direct classification ap-

proach and an LSTM model apart from our study. One of the differences between our study and [64] is that they take opcodes as basic units, whereas we take opcodes and operands together as instructions. Also, we created two datasets for two different data representations of applications to be forwarded to our LSTM model.

This thesis suggests an unsupervised learning approach by utilizing NLP methods. Hence, our model does not require feature extraction and laborious pre-processing, making it a more practical candidate. We employed the LSTM model for detecting obfuscation methods. We trained the model with the instruction sequence (ISD) and the method sequence (MSD) representations of applications. Our approach to dealing with the obfuscation detection problem with NLP techniques will be detailed next. We will also explain how we built our datasets and the architecture and parameters of our model in the Methodology section.

Deobfuscation

We mentioned that obfuscation is a popular and effective technique to protect Android applications against reverse engineering, and both application developers and malware authors widely adopt obfuscation. The widespread usage of obfuscation introduces many challenges in the reverse engineering process, analysis of applications and malware, and security evaluation of third-party libraries. Therefore we mentioned how crucial it is to detect obfuscated applications and presented the obfuscation detection studies in the literature. Moreover, defeating the obfuscation techniques is as crucial as detecting them. Thus, the deobfuscation process is the next step after detecting an obfuscation method. Even though it is out of the scope of our study, it is essential to mention deobfuscation to give a holistic view of obfuscation. Deobfuscation is the opposite of the obfuscation process. Deobfuscation aims to simplify and remove the obfuscated code to make an application's source code more understandable. Authors in [111] state that deobfuscation focuses on three key factors to understand a program better: readability, a better understanding of the program's control and data flows, and getting context.

Several automated deobfuscation solutions have been developed to make reverse engineering analysis of Android applications easier. In [112], the authors presented a string deobfuscation method to obtain decrypted strings through dynamic analysis. In another study [113], researchers suggested a tool called DeGuard that focuses on deobfuscating the identifier and package renaming methods of ProGuard by utilizing a probabilistic learning model. The authors stated that their model could defeat identifier renaming and detect third-party libraries, and their proposed tool can aid analysts in examining obfuscated malware. Moreover, in the study [114], the authors proposed a tool that can recover debug information using machine learning. Lastly, the researcher in [115] offered an automated system that can recover the original control flow graph of an obfuscated program by Obfuscator-LLVM.

2.7 Summary

In this section we provided the necessary background information for the concepts related to this thesis. We started with explaining the architecture of Android ap-

plication package and the difference between Dalvik bytecode and smali code. We disassembled Android applications and obtain their smali codes to use in our study. Then we explained and defined the obfuscation in general. We focused on obfuscation in Android and presented different obfuscation methods from the literature under four categories as trivial, non-trivial, preventive, and custom techniques. Also, we provided the available commercial and open-source Android obfuscation tools. Afterward, we presented the parallel developments and trends in detection methodologies including malware, clone, and third-party library detection. We started with signature-based methods and stated their limitations. Then we described ML-based approaches. After stating the limitatinos of ML-based detection methodologies, we provided necesssary information on neural networks, especially RNN and LSTM. We also explained NLP and the advancements in NLP. Then we explained detection methodologies that utilized DL approaches. Finally, we reviewed the literature on Android obfuscation detection and presented the related studies.

CHAPTER 3

METHODOLOGY

In this section, our methodology will be detailed by describing our approach to dealing with the obfuscation detection problem in the Android environment. Then we will describe how we built our dataset and explain the applied obfuscation techniques. Lastly, we will present our LSTM architecture for detecting obfuscation.

3.1 Scope and Approach

Obfuscation detection methods show a similar historical development to malware detection methods and utilize static, dynamic, and hybrid analysis techniques. Static analysis is the process of reviewing and evaluating an application by looking at the manifest file or source code without executing the application. On the other hand, dynamic analysis refers to examining an application by executing it and observing its behavior and outcome during runtime. There are also hybrid analysis techniques that utilize and combine both the static and dynamic analysis methods. Static and dynamic analysis techniques can be considered complementary, and each approach has its own advantages and disadvantages regarding needed resources, accuracy, and time. For example, an application must be operated in a separate, isolated environment to perform dynamic analysis. Moreover, dynamic analysis is a more complex procedure that can be resource and time-consuming in some cases. Since we are focusing on detecting static obfuscation methods, we applied the static analysis technique due to its simplicity and high effectivity in this thesis. Also, static analysis investigates all potential execution paths and variable values instead of only the ones invoked during runtime, making it a more suitable choice for obfuscation detection.

Early obfuscation detection approaches utilized signature-based methods. The signatures were crafted mainly from popular obfuscators and their specific configurations. Signatures can be generated from some specific strings or implementation of the obfuscation transformation (e.g., renaming policy, preferred encryption algorithm for constant strings). A significant challenge has been that numerous obfuscation techniques and tools are accessible, so one may customize and implement a novel obfuscation technique that would be difficult to detect with available signatures. Moreover, generating a new signature is time-consuming, and it takes significant manual effort, in contrast to relatively little effort to avoid detection. Consequently, traditional signature-based obfuscation detection methods became ineffective. Considering the vast number of Android applications and malware in the wild, the need for automated systems became essential for obfuscation detection, similar to malware and clone de-

tection. Machine learning approaches to detection problems have allowed the development of automated systems that outperform traditional, signature-based methods. Classical machine learning techniques have certain limitations due to the complexity of feature selection and extraction processes. Nevertheless, deep learning approaches allow the development of high-performing models for obfuscation detection. In the present study, we utilized static analysis techniques and Recurrent Neural Networks (RNN) to overcome the limitations of signature-based and classical machine learning methods. Our model does not require signature generation, feature extraction, or time-consuming pre-processing.

In this thesis, we are inspired by the work in [26]. The authors tackled the malware detection problem by utilizing NLP methods. Researchers collected run trace outputs of the malware and benign Portable Executable (PE) files for Windows OS. Afterward, by taking advantage of the similarities between natural languages and assembly language, they modeled benign and malware files as if they were two different languages. Then, they used an LSTM model for the binary classification of malware and benign samples. The resulting models obtained outstanding results in detecting malware. The study [26] showed that applying NLP methods to programming languages is a sensible approach, and programs with different syntax and semantics can be modeled as different languages in some cases. One of the interesting points that inspired us in the work [26] is the analogy between natural languages and assembly language. In natural languages, a word is the basic and nominative unit. Words come together structurally and contextually to form sentences and enable us to express basic statements. Then sentences group together and compose paragraphs to express more complex meanings, intentions, instructions, or any information. The same analogy also holds for smali language, which consists of several units such as instructions, basic blocks, methods, and classes. The basic units in smali are the opcodes and operands, which correspond to words in natural languages. Opcodes and operands form instructions. They have machine-interpretable functionality, meaning, and dependency, akin to articulating a proposition using sentences as in natural languages. Likewise, instructions form basic blocks, basic blocks form methods, and methods constitute classes. Hence as we go up from instructions to classes, expressed functionality and meaning get more complex, and contextual information increases, similar to natural languages. In line with this information, leveraging NLP techniques gives the opportunity to grasp some specific patterns, the intention and behavior of the code.

In our case, the problem is to detect whether a given Android application code is obfuscated or not with the chosen method. In a basic sense, implementing obfuscation can be seen as a protection mechanism against reverse engineering. Some studies categorize obfuscation as a measure against man-at-the-end attacks [46]. These attacks are performed by end-users who have complete control over the released product, an Android application in our case. The end-users can be both malicious actors and security analysts. They perform various analysis techniques to reverse the application and obtain its logic. However, we will look at obfuscation in a man-in-the-middle attack scenario in this thesis by focusing on communication. This scenario can be thought of as follows: There is a communication between a developer and a computer through a code piece. The attacker is an analyst or a malicious actor who wants to reverse this communication to understand what the developer tells the computer via

source code. The role of the obfuscation in this scenario is to modify the code syntactically in a way to make the communication inarticulate for the attacker for more challenging analysis and comprehension of the code's behavior while preserving the original semantics, hence the original functionality. Nevertheless, some obfuscation methods may introduce some side effects and additional functionality that can change the semantics of the application due to inserted junk code and concealed malicious payloads. Therefore, we wanted to model original and obfuscated samples for each chosen obfuscation technique as if they were different languages by utilizing NLP techniques as in study [26]. We wanted to investigate if it is feasible to model smali language obtained from disassembled Android applications using a special type of neural network, LSTM, in obfuscation detection.

Obfuscation techniques include a wide range of transformations that maintain the original semantics and functionality of the application. In [116], researchers reviewed obfuscation algorithms and stated the general characteristics of the obfuscated codes. Obfuscated codes have structural characteristics due to the implementation of many loops and nesting. Moreover, due to the examples such as variables defined at the beginning of a code can be reused at the end, obfuscated codes have long dependencies. Therefore, the chosen neural network for obfuscation detection has to handle this structural and contextual information and has to have a broader context window to gather maximum semantic information while keeping the sequence order to detect specific patterns and dependencies. Similar requirements are also applicable for detecting obfuscation methods in the Android environment. Therefore, to meet the needs of our study, we utilized Long Short-Term Memory (LSTM), a special kind of Recurrent Neural Networks (RNN) architecture. LSTM architectures have better results at solving vanishing and exploding gradient problems and are better candidates for extracting semantic information from sequential data. Since all sorts of transformations can be applied to the code counted as obfuscation, and there are various commercial and open-source obfuscators for the Android environment, it is not possible to consider every obfuscation method and tool. Therefore, we used an open-source obfuscator obfuscapk [77] which is suitable for automation and designed for academic research. We chose several widely-used obfuscation methods available on obfuscapk and created our own obfuscated samples. Then we employed a static analysis approach and disassembled original and obfuscated applications by using apktool [31]. We used different representations of the smali language and created two datasets to train the LSTM model. At first, we represented collected applications as sequences of instructions and forwarded these sequences to LSTM for detecting obfuscation techniques. Secondly, applications are represented as sequences of methods. According to our results which will be detailed later on, the second representation outperforms the first one as expected since methods have more complex functioning and meaning than instructions.

3.2 Overview of the Proposed Methodology

Our proposed methodology for detecting obfuscation consists of disassembling, obfuscation, light preprocessing, and training/testing stages. In the disassembling stage, we obtain the source code of each original Android application in the Smali language

and write the smali code to a text file with the name of the corresponding application. In the obfuscation stage, we first obfuscate each apk with nine obfuscation techniques separately. Then we follow a similar procedure as in the disassembling stage and write the source code of obfuscated applications into text files named with the name and the applied obfuscation method of applications. Before the training/testing stage, we perform a light preprocessing on the previously generated text files to represent them as sequences of instructions and sequences of methods models. Afterward, the two representation models are forwarded to the LSTM architecture as vectors after tokenization and word embedding. Lastly, in the final training/testing stage, we train the LSTM model for three epochs to decide whether the application is obfuscated or not with binary classification. We will explain the implementation details of disassembling, obfuscation, and light preprocessing stages in the Datasets section. Then, the implementation details of the training/testing stage will be given in the LSTM Model section. The overview of our proposed system and modeling pipeline for each detection unit can be seen below in Figure 3.1 and Figure 3.2, respectively.

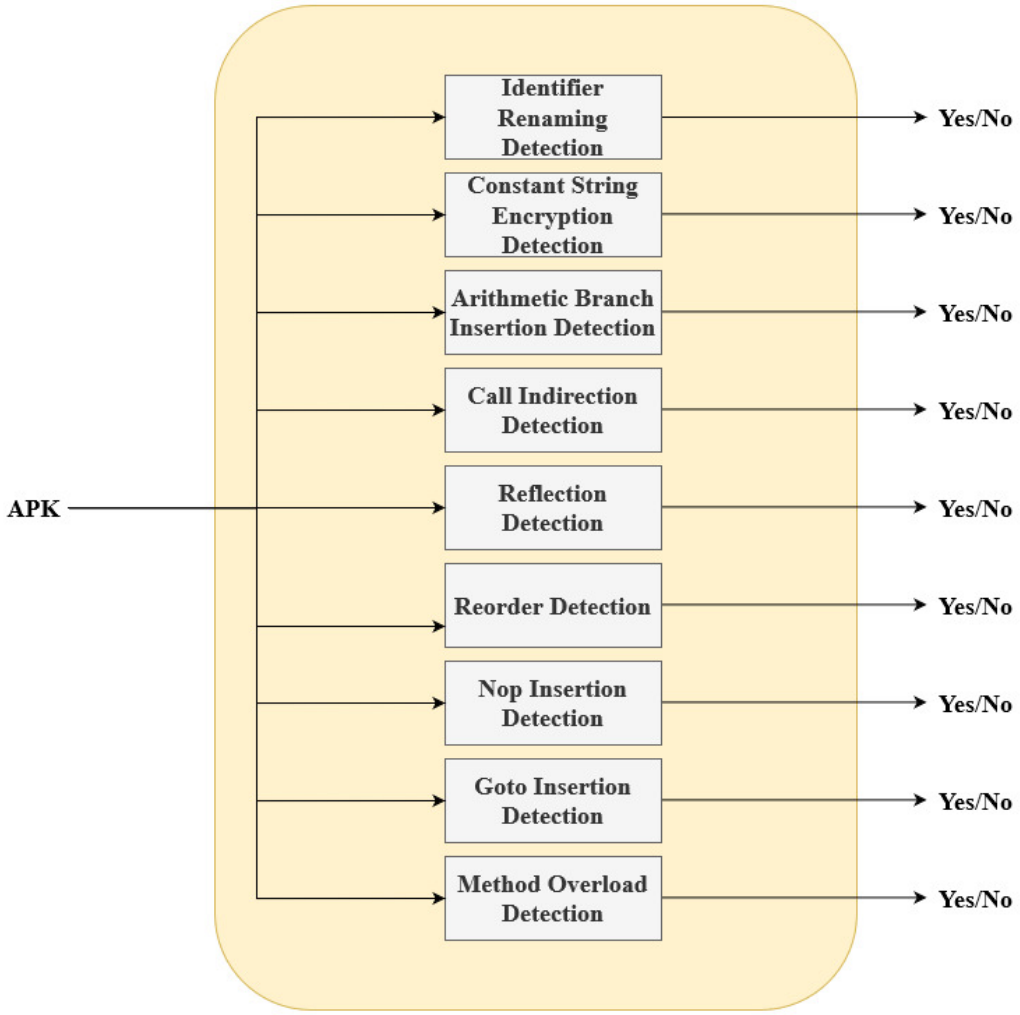


Figure 3.1: The overview of the methodology in the present study

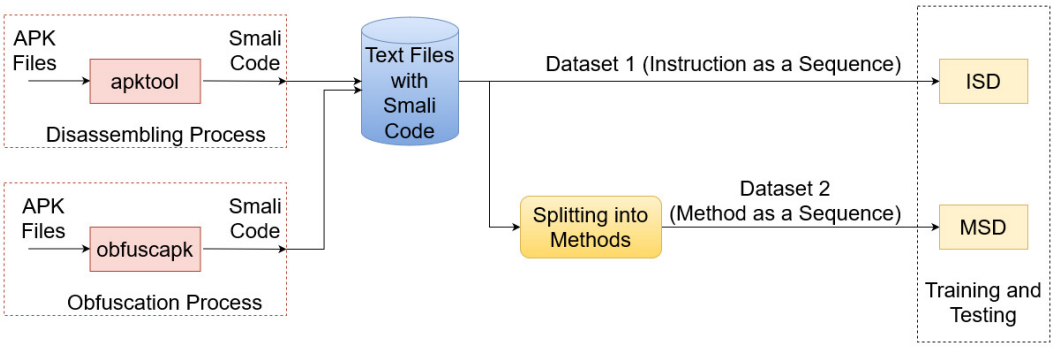


Figure 3.2: The processing pipeline for each detection unit in the model

We should also point out that we analyzed the detection of each obfuscation method individually and developed binary classification-based LSTM models. We wanted to investigate the feasibility of modeling a chosen obfuscation method individually and observe the success of the LSTM model in detecting different obfuscation methods. Therefore, the obfuscation process shown in Figure 3.2 is repeated nine times for each implemented obfuscation technique.

3.3 Datasets

Since obfuscation detection is not a widely studied subject and there are many different obfuscation techniques in the wild, we could not find a public dataset that would meet our needs for the study. Therefore, we built our datasets[1] for training the model. To create the datasets, we downloaded 324 applications from the F-Droid [117] website. The reason we chose F-Droid is due to its open-source nature. We mentioned that obfuscation is employed by both malware authors and legitimate software developers. Malware authors use obfuscation to evade the detection engines, and legitimate software developers use it to protect their intellectual property and economic gain. We purposely did not include malware samples and commercial applications in the dataset since they may already be obfuscated and cause errors during training the model. After the download process, apks are disassembled and copied into text files with a Bash script to form the non-obfuscated samples as part of the disassembling stage of our methodology. The script uses apktool [31] to disassemble the apks as a first step. apktool is an open-source reverse engineering tool that allows decompiling and recompiling Android applications. According to their website [31], it can decode resources to the nearly original form and rebuild them after applying some modifications and transformations, which makes it a suitable choice for working with obfuscation. Afterward, smali directories created by apktool are recursively traversed by the script, and classes of applications are copied to text files. This process results in copying the source code of each apk to its corresponding text file with the same name of the apk. The resulting 324 text files consist of the smali language. At the end of this process, we obtain our original samples as text files which will be used in every obfuscation detection model.

[1] We will share the dataset upon request on https://github.com/bulukapi/AndrObfuscationSet

To obfuscate the apks, we used an open-source tool obfuscapk [77] that was developed by researchers from the University of Genoa. obfuscapk provides various options for obfuscation and is also integrated with our choice of a disassembler, apktool. We selected and implemented the following obfuscation methods to each collected apk:

- **Identifier Renaming**:

  The obfuscator supports several options to change the name of the identifiers used in the code. We implemented the field, method, and class renaming techniques altogether. Therefore, the resulting obfuscated applications contain modifications in all these three identifiers. This method changes all field, class, and method names to nine character long random strings. The effect of the identifier renaming obfuscation on the code can be seen in Figures 3.3 and 3.4. The identifiers in Figure 3.3 get transformed into random strings that can be seen in Figure 3.4.

```
.method public d(Landroid/os/Bundle;)V
    .locals 1

    iget-object v0, p0, Landroidx/savedstate/a;->b:Landroidx/savedstate/SavedStateRegistry;

    invoke-virtual {v0, p1}, Landroidx/savedstate/SavedStateRegistry;->c(Landroid/os/Bundle;)V

    return-void
.end method
```

Figure 3.3: A sample code before identifier renaming obfuscation

```
.method public d(Landroid/os/Bundle;)V
    .locals 1

    iget-object v0, p0, Lp576f3918/pc9f3ee6d/p0cc175b9;->f92eb5ffe:Lp576f3918/pc9f3ee6d/p8acb42d4;

    invoke-virtual {v0, p1}, Lp576f3918/pc9f3ee6d/p8acb42d4;->c(Landroid/os/Bundle;)V

    return-void
.end method
```

Figure 3.4: A sample code after identifier renaming obfuscation

- **Constant String Encryption**:

  One of the obfuscation techniques provided by the tool is constant string encryption. obfuscapk generates a 32-bit random key using ASCII letters and digits on startup to use for encryption later on. The tool uses the AES algorithm in ECB mode for constant string encryption. In terms of cryptology, using a key less than 128 bits long and using AES in ECB mode does not provide security. An analyst can easily obtain the secret key to decrypt the strings in the obfuscated code. Therefore even though the method is called encryption, it is not encryption in the true sense, and it can be categorized as an obfuscation technique. The tool searches for constant strings in the code and encrypts them according to the encryption method explained previously. As we can see from Figures 3.5 and 3.6, constant strings "context.getString(nameRes)" and "Category(id=" in

Figure 3.5 becomes "f78dff6fe6b875c01ed6531ca8046223a3f2573e485d4e125 2bd202260741f1c" and "8697f21332fe584445d437a6178418ba" after applying the obfuscation method.

```
invoke-virtual {p1, v0}, Landroid/content/Context;->getString(I)Ljava/lang/String;

move-result-object p1

const-string v0, "context.getString(nameRes)"

invoke-static {p1, v0}, Lkotlin/jvm/internal/Intrinsics;->checkExpressionValueIsNotNull(arguments;)V

invoke-direct {v1}, Ljava/lang/StringBuilder;-><init>()V

const-string v2, "Category(id="

invoke-virtual {v1, v2}, Ljava/lang/StringBuilder;->append(arg;)Ljava/lang/StringBuilder;
```

Figure 3.5: A sample code before constant string encryption

```
invoke-virtual {p1, v0}, Landroid/content/Context;->getString(I)Ljava/lang/String;

move-result-object p1

const-string/jumbo v0, "f78dff6fe6b875c01ed6531ca8046223a3f2573e485d4e1252bd202260741f1c"

invoke-static {v0}, Lcom/decryptstringmanager/DecryptString;->decryptString(arg;)Ljava/lang/String;

move-result-object v0

invoke-static {p1, v0}, Lkotlin/jvm/internal/Intrinsics;->checkExpressionValueIsNotNull(arguments;)V

invoke-direct {v1}, Ljava/lang/StringBuilder;-><init>()V

const-string/jumbo v2, "8697f21332fe584445d437a6178418ba"

invoke-static {v2}, Lcom/decryptstringmanager/DecryptString;->decryptString(arg;)Ljava/lang/String;

move-result-object v1
```

Figure 3.6: A sample code after constant string encryption

- **Arithmetic Branch Insertion**:

  This obfuscation method is similar to other junk code insertion methods. The added junk code in this type of obfuscation is constructed with arithmetic computations and a branch instruction. The branch instruction is never taken due to the arranged computations. Also, if a method in the application code has at least two available registers, the tool adds a fake branch at the beginning of the method. Then by using “goto” instructions, one branch will go to the end of the method first and then go to the fake branch created at the beginning of the method. This transformation and the added code can be seen in Figure 3.7 below.

- **Call Indirection**:

  This method changes the control-flow graph while preserving the semantics of the code. In this obfuscation technique, proxy methods that call the original methods are added to the code. As shown in Figure 3.8, proxy methods have randomly generated names and are used to invoke the original methods.

```
.method public a(Ljava/lang/Object;Ljava/lang/Object;)Ljava/lang/Object;
    .locals 3

    const v0, 16
    const v1, 16
    add-int v0, v0, v1
    rem-int v0, v0, v1
    if-gtz v0, :XMnRjuwkTAXfyFmV
    goto/32 :oywZXwRaaQtiMaaE
    :XMnRjuwkTAXfyFmV
    :qoGMmfNgTAepDuEg

    --here is the original code--


    :oywZXwRaaQtiMaaE
    goto/32 :qoGMmfNgTAepDuEg
.end method
```

Figure 3.7: Added code in arithmetic branch insertion

```
.method public static dyybLGGePqyYVriD(I)Ljava/lang/String; //added proxy method

    invoke-static {p0}, Ljava/lang/Integer;->toHexString(I)Ljava/lang/String; //original method

    move-result-object v0

    return-object v0
.end method

.method public static PZzunplMSqVxoBpE(Ljava/lang/Object;)I //added proxy method

    invoke-static {p0}, Ljava/lang/System;->identityHashCode(Ljava/lang/Object;)I //original method

    move-result v0

    return v0
.end method
```

Figure 3.8: Added proxy methods in call indirection

- **Reflection**:

  This obfuscation technique is specific to the Java language and similar to the Call Indirection technique. Instead of adding new methods to invoke the original ones, this technique uses Reflection API as a proxy and calls original methods through the API. For reflection obfuscation, the tool needs to search for proper method invocations in the code. According to [77] and their source code, a method must have the following properties to get obfuscated by the reflection technique:

  - ➢ It is not a part of the Android Framework.
  - ➢ It has a public visibility.
  - ➢ It is not a constructor method.
  - ➢ There are enough free registers for remaining method invocations.

  The methods satisfying the above properties can be obfuscated by Reflection technique.

- **Reorder**:

  The reordering technique aims to change the ordering of the basic blocks in the code. The tool searches for branch instructions in the code and inverts the conditions of branches. For example, “if-eq” is inverted to “if-ne”, “if-gtz” is

inverted to “if-lez” and so on. Moreover, the tool also reorganizes the code by adding goto instructions inside the method implementations. This technique also inserts nop instructions into the code. The effects of the reordering technique on the code can be seen in Figure 3.9.

```
.method private constructor <init>(arg;)V
    .locals 0

    goto/32 :l_asWdaXlclAmErrsH_0

    nop

    :l_JlQrXGOXkJrKNwla_2
    return-void
    :l_asWdaXlclAmErrsH_0
    iput-object p1, p0, Lcom/material/floatingactionbutton/b$f;->b:Lcom/material/floatingactionbutton/b;

    goto/32 :l_rRiUOWOnKjxHBbJL_1

    nop

    :l_rRiUOWOnKjxHBbJL_1
    invoke-direct {p0}, Landroid/animation/AnimatorListenerAdapter;-><init>()V

    goto/32 :l_JlQrXGOXkJrKNwla_2

    nop

.end method
```

Figure 3.9: A sample code after reorder

- **Nop Insertion**:

  Nop Insertion is the most basic junk code insertion technique. In this obfuscation method, the added junk code is the nop instruction which does nothing. The tool inserts random nop instructions inside every method implementation.

- **Goto Insertion**:

  This obfuscation technique aims to modify the control-flow graph by adding goto instructions. The tool adds two goto instructions, one in the beginning and one at the end of each method. If we follow the control flow in Figure 3.10, the first goto points to “after_last_instruction”, after following that second goto points back to “before_first_instruction”, so that the flow can be passed to the original code. Adding goto instructions in this way also adds two new nodes to the control-flow graph.

```
.method public static varargs addQueryParameters(args;)Ljava/lang/String;
    .locals 4

    goto/32 :after_last_instruction

    :before_first_instruction


    --here is the original code--


    :after_last_instruction

    goto/32 :before_first_instruction

.end method
```

Figure 3.10: Added code in goto insertion

- **Method Overload**:

  Java language has a feature called method overloading. This feature allows performing one or more functions with the same method name, but the methods have to differ in used arguments. The overloading functionality has the advantages of increasing performance and saving from memory by providing the reusability of the program. On the other hand, it can be exploited as an obfuscation technique. The tool employs the method overloading technique to create a new method with the same name as the given method and adds new random arguments to the method. Then the tool fills the method's body with random arithmetic instructions, as shown in Figure 3.11.

```
.method public static a(IIBCZF)V

    const/16 p0, 0x2a
    const/16 p1, 0xd2
    mul-int p2, p0, p1
    add-int p3, p2, p1
    int-to-double p0, p3

    return-void
.end method

.method public static a(IIZCFB)V

    const/16 p0, 0x2a
    const/16 p1, 0xd2
    mul-int p2, p0, p1
    add-int p3, p2, p1
    int-to-double p0, p3

    return-void
.end method
```

Figure 3.11: Added code in method overload

Each apk is obfuscated with previously mentioned methods separately in the obfuscation process. Hence, we obtained nine different versions of each application per obfuscation technique, resulting in 2916 apks for datasets. To disassemble and copy the source code of obfuscated apks to text files, we go through a similar process to creating the original samples. Since obfuscapk decompiles apks with apktool and outputs their smali directories as an initial step during the obfuscation process, we modified our initial Bash script to obfuscate applications and obtain their text file representations. The modified script uses obfuscapk to obfuscate and obtain the smali files of the obfuscated applications. Like the original script, smali directories are traversed recursively, and classes of applications are copied to text files with the name of the application and implemented obfuscation method. In the end, we have 3240 text files corresponding to each original and obfuscated apks for our datasets.

In this thesis, we created two datasets from the generated text files with Smali code. The difference between these two datasets is the representations of the applications. In the first dataset, namely Instruction Sequence Dataset (ISD), we represented applications as instruction sequences, meaning every line in the text files corresponds to an instruction. We performed a simple preprocessing using a Python script to form the ISD. The script starts reading the source code of applications from the text files line by line. When it encounters a line starting with '.method,' it saves the lines until a line with '.end method' is seen. However, we filter out specific lines before saving only

to get the instructions and then write the instructions per line. The filtered content can be stated as follows: comment lines which start with '#', lines starting with an end-line character and ':', and the lines starting with '.registers', '.locals', ',prologue' and '.line'. In this way, we obtain our instruction sequence dataset where every line is an instruction. Sample lines from the ISD can be seen in Figure 3.12.

```
sget-object v0, Lcl/MainActivity$SourceChange;->CHANGE_FROM_MAP:Lcl/MainActivity$SourceChange;
if-eq p1, v0, :cond_1
sget-object p1, Lcl/coders/faketraveler/MainActivity;->editTextLng:Landroid/widget/EditText;
invoke-virtual {p1}, Landroid/widget/EditText;->getText()Landroid/text/Editable;
move-result-object p1
invoke-static {p1}, Ljava/lang/Double;->parseDouble(Ljava/lang/String;)D
move-result-wide v0
const-string v2, "INPUT"
const/4 v3, 0x1
```

Figure 3.12: Sample lines from the ISD

Then in the second dataset, namely Method Sequence Dataset (MSD), we represented applications as method sequences. The procedure for building the MSD is very similar to the ISD. Hence, we modified the initial Python script. Even though the script works the same, the difference is that the new script does not write the instructions between '.method' and '.end method' line by line. Instead, it saves them as a sequence, and when it sees the line starting with '.end method', it writes the method sequence as a line. We can see the sample lines of the MSD in Figure 3.13.

```
const-string v0, "viewModel"     invoke-static {p1, v0}, Lkotlin/jvm/internal/
Intrinsics;->checkNotNullParameter(Ljava/lang/Object;Ljava/lang/String;)V     sget v0, Lademar/bitac/
R$id;->root:I     invoke-virtual {p0, v0}, Lademar/bitac/view/
CheckAddressActivity;->_$_findCachedViewById(I)Landroid/view/View;     move-result-object v0
check-cast v0, Landroid/widget/RelativeLayout;     new-instance v1, Lademar/bitac/view/
CheckAddressActivity$showSave$1;     invoke-direct {v1, p0, p1}, Lademar/bitac/view/
CheckAddressActivity$showSave$1;-><init>(Lademar/bitac/view/CheckAddressActivity;Lademar/bitac/
viewmodel/WalletViewModel;)V     invoke-virtual {v0, v1}, Landroid/widget/RelativeLayout;->post(Ljava/
lang/Runnable;)Z     return-void

invoke-direct {p0}, Landroid/preference/PreferenceFragment;-><init>()V     return-void

iget v0, p0, Lademar/bitac/view/Theme;->resTheme:I     return v0

invoke-virtual {p0}, Lademar/bitac/view/CheckAddressActivity$onCreate$8;->invoke()V     sget-object
v0, Lkotlin/Unit;->INSTANCE:Lkotlin/Unit;     return-object v0

new-instance v0, Lademar/bitac/view/WalletAdapter$add$2;     invoke-direct {v0}, Lademar/bitac/view/
WalletAdapter$add$2;-><init>()V     sput-object v0, Lademar/bitac/view/
WalletAdapter$add$2;->INSTANCE:Lademar/bitac/view/WalletAdapter$add$2;     return-void
```

Figure 3.13: Sample lines from the MSD

We created the ISD and the MSD to detect all nine obfuscation techniques separately. The resulting total sequence numbers of the Instruction Sequence Dataset (ISD) and Method Sequence Dataset (MSD) per obfuscation method can be seen in Table 3.1 below. We should also mention that the source code of the disassembled applications is stored in separate text files before creating the ISD and the MSD. However, while building the datasets, we collected all the content of the separate files into two text files for each obfuscation method detection. The reason behind this choice is to increase the performance by getting rid of the burden of opening and closing each text file to read data during training and testing the model. For example, to detect renaming obfuscation, we collected all the source code of original apks into a text file

called “NonObfuscated” and the apks obfuscated with renaming into a text file called “RenamingObfuscated”. Then we trained the model using the two merged text files. The exact process is applied to each chosen obfuscation detection unit.

Table 3.1: The characteristics of the two datasets (ISD and MSD)

	Number of Sequences (millions)	
	Instruction Sequence	Method Sequence
Original	141	7.5
Identifier Renaming	141	7.5
Arithmetic Branch Insertion	168	7.5
Call Indirection	160	15.0
Goto Insertion	172	7.6
Method Overload	157	10.0
Nop Insertion	200	7.5
Reflection	156	7.6
Reorder	200	7.6
Constant String Encryption	149	8.6

3.4 LSTM model

The last stage of our proposed methodology is the training/testing stage. In this section, we will provide the technical details of our setup environment and neural network model. We will also describe the working pipeline and parameters for training and testing the model. We will start this section by explaining the advantages of the LSTM architecture for obfuscation detection problem.

Deep neural networks have recently gotten much attention in areas including speech recognition, image processing, and natural language processing. Many studies employ NLP techniques on programming languages due to the similarities between natural languages and programming languages, as we previously mentioned. Specifically, the LSTM model is employed by numerous NLP-based malware detection studies and performed successfully for detecting patterns and behavior of malware from instruction, opcode, and API sequences. The LSTM model proved that it is a good choice that can learn the behaviors and intentions of long sequences. We also know that LSTM is better at solving vanishing and exploding gradient issues than other Deep Neural Network architectures. Similar to malware, implementing obfuscation introduces some specific characteristics and dependencies to the code. First of all, obfuscated code mainly includes a lot of loops, conditional statements, and nesting that are critical indications for detecting the obfuscation method. Moreover, obfuscation is closely related to the context, which may have long dependency intervals. For instance, for detecting control-flow obfuscations model should be able to track long dependencies in sequences. In another example, an identifier renamed at the beginning of a code may be used through the end. Hence, the model has to deal with long sequences efficiently. In line with these characteristics of obfuscated code, the LSTM

model is a suitable candidate for detecting obfuscation. The existence of forget input, and output gates in LSTM, enables the model to grasp specific patterns and behavior from long sequences. In this study, we employed an LSTM architecture for modeling the smali code of Android applications in order to grasp the semantic relation of sequences to detect obfuscation.

3.4.1 Environment Setup

We performed training and testing of our proposed models for this study at TUBITAK ULAKBIM, High Performance and Grid Computing Center (TRUBA resources). We used "barbun-cuda" computing set, which has an Intel Xeon Scalable 6148 CPU and NVIDIA Tesla P100 GPU. The server we used had Centos Enterprise Linux 7.3 operating system [118]. To develop our LSTM model, we used Python version 3.6.5 as our programming language. We also employed several modules and libraries in Python for model development which will be detailed below. The used libraries and their versions are also specified in Table 3.2 below.

Table 3.2: The used libraries and their versions to build the LSTM model

Tensorflow	2.1.0
Keras	2.2.4
Pickles	4.0
Scikit-learn	0.24.1
Numpy	1.19.5
Matplotlib	3.3.4
Seaborn	0.11.1

The imported libraries and modules to build and train the LSTM model and their purposes are briefly explained as follows:

- **Os**: Os is a built-in Python module that can perform operating system tasks for working with directories and files. Since os is a built-in module, it follows the version of Python, which is 3.6.5. We use this module to list directories to access the merged files and read their contents. The module is also used while labeling the data before training the model.
- **Tensorflow**: Tensorflow is a popular ML framework that can be used for employing ML and DL applications. It also includes the Keras library, which is used for deep learning, specifically for neural networks. Keras library can be considered as an interface to Tensorflow. We trained our LSTM model using Tensorflow and Keras libraries. The imported modules from these libraries can be stated as follows:
  - **confusion_matrix**: This module is imported from Tensorflow's math library. We used this module to compute the confusion matrix from the given test set and predictions.

  - **Keras**: As we mentioned previously, the Keras library is also imported from Tensorflow. Keras provides several functional modules that enable us to build our LSTM model. We imported the modules below from the Keras library.
  - **Tokenizer**: Tokenizer module is imported from keras.preprocessing.text and provides text tokenization. We used this module to vectorize and encode our text corpus of instruction sequences into sequences of integers.
  - **pad_sequences**: The module is imported from the keras.preprocessing.sequence and used to pad or truncate the sequences into our fixed sequence length that we use to train the model.
  - **Input, Embedding, LSTM, Bidirectional, Dense, GlobalMaxPool1D, Dropout, Activation**: All of these modules are imported from Keras' layers module. These modules are used to build the layers of our LSTM architecture. We will explain the details of the layered architecture of our LSTM model in Section 3.4.2.
  - **Sequential**: Sequential module is imported from Keras' Models library. This module gathers a linear stack of layers into a Model object and allows us to train and work on the model.
  - **Adam**: We imported this module from Keras' Optimizer library. It implements the Adam algorithm as the optimizer needed to train the LSTM model.

- **Numpy**: It is a widely used library for scientific computations that helps developers save a lot of time in scientific computations that need complex matrix operations. Numpy library provides a particular class of arrays called Numpy arrays that perform vast matrix-based calculations efficiently. It is also integrated with TensorFlow for manipulating tensors at the backend. In the present study, we utilized Numpy arrays to divide the dataset into train, test, and validation splits. The data stored in python lists are converted to Numpy arrays and forwarded to train and test the model during the splitting process.

- **Pickles**: Pickles is a default module of the Python library that allows sending complex object hierarchies over a network or saving the internal state of objects to a disk or a database through pickling and unpickling. Objects are converted to byte streams during the pickling process, and the inverse operation is performed while unpickling. After we label our data in our study, we build the model's vocabulary and tokenize the text data. After the tokenization process, we save our tokenizer using the Pickle module in case of future use. In this way, we avoid repeating the tokenization process for every trial.

- **Scikit-learn**: Scikit-learn is a popular tool used for building ML algorithms and data preprocessing and modeling. It provides various data preprocessing features, and once the data is preprocessed, it can be quickly passed into TensorFlow.

  - **train_test_split**: In the present study, we imported train_test_split module from Scikit-learn's Model_selection. We used the module to separate the dataset into train, test, and validation partitions. The details about this split will be detailed in Section 3.4.2.

- **Matplotlib**: Matplotlib is a popular library used for data visualization and plotting.
  - **pylot**: From the Matplotlib library, we imported pyplot module to use in our study. We used pyplot to save the created confusion matrix. Also, it is utilized the plot and saves the accuracy and loss graphs for each trained model.
- **Seaborn**: Seaborn is a data visualization library based on matplotlib and used for statistical visualizations. We used this library to create graphical confusion matrices with heatmap from the confusion matrix created by Tensorflow.

3.4.2 Training/Testing

This section will introduce the working pipeline of the present study to train and test the LSTM model. As explained in the Environment Setup section, we build our neural network model using Tensorflow and Keras libraries. We modified the algorithm in [26] and [119] to train the LSTM model for detecting obfuscation from the Smali code of the Android applications. We will continue our discussion with the details of the training/testing stage of our methodology which is similar to study [119].

In Section 3.3, we explained how we prepared our datasets from the Smali code of the disassembled applications and modeled them in two different forms of representation. We also mentioned that we collected all text files into two merged files to increase the performance. The pipeline for training the LSTM model starts by opening the two merged text files. Merged text files contain original and obfuscated samples separately. Hence, in the next step, we read the contents of the merged text files line by line, which corresponds to instructions, and label them as '1' or '0'. Lines from the 'Nonobfuscated' file are labeled as '0' to indicate that they are not obfuscated, and lines from the 'ObfuscationNameObfuscated' file are labeled as '1' to indicate the obfuscation. We used two Python lists to store the data and the corresponding label during data labeling.

After labeling the data, we continue with building the dictionary for the model. To accomplish that, we have to tokenize the sequences first. We used the Tokenizer module from the Keras library for the tokenization process and used the word tokenization type. According to the Tokenizer Documentation [120], all punctuation is removed by default during tokenization, and then the texts are converted into space-separated sequences of words. Later on, these sequences of words are divided into lists of tokens. To exemplify the process from our study, let us consider the following sequence:

➡ check-cast v1, Ljava/lang/Boolean;

Tokenizer module first removes all the punctuation characters and outputs the following sequences of words:

➡ check cast v1 Ljava lang Boolean

Then, these sequences are divided into tokens in a specific order and stored in Python lists. After the tokens are created, we employ the fit_on_texts method of the Tokenizer module to update the internal vocabulary for the texts list. This method is necessary for us to encode our tokens later on. In this way, our tokenization process is completed, and our dictionary for the model is built. Before continuing the padding and encoding operations which is the next step of our pipeline, we save our resulting tokenizer by using the Pickle module for later use, without the need to repeat the tokenization process. While working with textual data, we need to convert the data into integers before feeding it into any machine learning model, including neural networks. The LSTM architecture also works on integer values for training. Since our tokens consist of strings instead of integers, we have to encode our tokens to be able to forward them to the LSTM model. Therefore, we use the texts_to_sequences method of the Tokenizer module, which converts tokens of text corpus into a sequence of integers. Each token gets assigned to a unique integer value during encoding except '0'. The index '0' is reserved and will not be assigned to any tokens. By doing so, we obtain a Python list including sequences of integer values.

The next problem is that all the integer sequences have different lengths; however, LSTM requires fixed-length sequences for training. We specify maximum sequence lengths of 10 for the ISD and 1000 for the MSD while training the model to deal with this issue. We also applied padding operations to the encoded sequences to satisfy this requirement. If a sequence's length is smaller than the maximum sequence length, then '0' is added at the end of the sequence until the length of the sequence reaches the maximum. We mentioned that '0' is reserved and does not correspond to any tokens; hence it is used in padding operations. On the other hand, if the length of a sequence is longer than the maximum length, we take the part of the sequence until the maximum length is reached and ignore the rest. With the end of the encoding and padding processes, we obtain a Python list that consists of fixed-length integer sequences.

After we prepared the data in the correct form to train the LSTM model, we have one more step before starting the training. Our next step is to divide the datasets into train, test, and validation splits for the training and testing of the model. Even though there are no specific rules about how to split the datasets, there is a standard convention followed by researchers. According to this convention, percentages of the splits are as follows: 60% of a dataset for training, 20% for testing, and 20% for validation. We also employed the standard convention in the present study and divided our dataset accordingly. We utilized the train_test_split method of the Scikit-learn library to split the datasets. Up to this point, we stored our data in Python lists, as we mentioned. Before forwarding the data to the model, we convert Python lists into Numpy arrays to increase the efficiency of the training process. After the conversion, our data is ready for training and testing the model. Now, we will explain how we built our LSTM model and give details about its parameters and configuration.

We wanted to observe the detection performance of the LSTM model on each obfuscation method separately. Therefore, we trained nine LSTM models with our datasets. The parameters and the architecture are the same for all nine models. The only difference is the dataset we used to train the models. On average, it took around 8 - 9 hours to train the model using ISD and about 1.5 - 2 hours to train using MSD, although the training time slightly differs for each obfuscation method. The LSTM model we

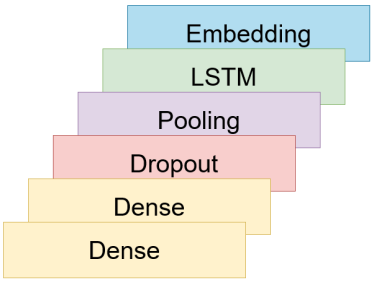


Figure 3.14: The layers of our proposed model

employed is similar to the one in the study [26]. Our model consists of six sequential layers, namely Embedding, LSTM, Pooling, Dropout, and two Dense Layers at the end as depicted in Figure 3.14. The Embedding Layer has mainly been used in NLP-related applications such as language modeling and converts positive integers into fixed-size dense vectors. In the Embedding Layer, we create the word embeddings using the dictionary we created in the previous steps. We specified the dimension of the dense embedding as 128 and input length as maximum sentence length (10 for ISD and 1000 for MSD). The second layer is the bidirectional LSTM layer. The bidirectional layer provides a bidirectional wrapper for RNNs. In this layer, we define our LSTM layer. Then the Pooling layer is used to lower the vector sizes. We used Global Max Pool 1D on this layer. Global Max Pool 1D takes the largest value across the time dimension to downsample the input representation. After the Pooling Layer, the Dropout Layer activates the selected nodes to increase the model's learning efficiency. Lastly, we used two Dense Layers at the end to reduce the dimensions from 128 to 64 and then from 64 to 2. Since we wanted to perform binary classification for nine obfuscation methods separately, we used a sigmoid as an activation function at the last layer. After building the LSTM architecture, we compile our model and start training. When the training is over, we save the weights of the current model for later use as we saved the tokenizer. Then by using Tensorflow's confusion_matrix method, we plot the confusion matrix of the model. We also plot the accuracy and loss graphs of the model and save them. Then we conclude our working pipeline by testing the model with test split. The confusion matrix of each obfuscation detection unit will be supplied in Appendix.

Parameters and Configuration

We performed several trials to determine the most suitable values for parameters such as maximum sentence length, dropout rate, and LSTM nodes for our model. For the ISD, we tried the values of 6, 8, 10, and 15. The maximum sequence length of 10 performed better than 6 and 8 in terms of loss and accuracy. When we increased it to 15, we did not observe considerable changes, and we chose the maximum sequence length of 10 for the ISD. Then for the MSD, we experimented with maximum sequence lengths of 100, 200, 500, and 1000. The sequence length of 1000 achieved the best results among other sequence lengths. Hence, we chose 1000 as the maximum sequence length for the MSD. For the dropout rate, we tried the values of 0.2, 0.5, and 0.8. After the trials, we chose the dropout rate as 0.2, considering the accuracy and loss rates. Lastly, we also observed the performance of the different numbers of LSTM nodes. We tried the values of 32, 64, 128, and 256. As we increased the

number to 64 from 32, the accuracy rate increased. As we continued to increase the number of LSTM nodes, accuracy increased, but after 64, we observed negligible improvements in the accuracy and loss. For all the parameters we tried, if the increase in accuracy rate is not significant, we preferred choosing the smaller value for the parameters. The reason behind this choice is to improve the training time. After the trials, our final model has the following parameters: the number of output nodes in the LSTM layer = 64; the dropout rate = 0.2; the pooling is Global Max Pool 1D; the first activation function is relu; the last activation function is sigmoid; the loss is binary cross-entropy; the optimizer is Adam with learning rate = 0.001; the batch size = 2048. We also observed that our models' accuracies stayed the same after 2 or 3 epochs. Hence we trained the models for three epochs.

3.5 Summary

In this section, we presented the scope of this thesis and our approach to detecting obfuscation methods by utilizing language modeling and the LSTM model. We proposed a method that performs binary classification on the smali code of the disassembled applications to detect whether an application is obfuscated or not. We stated the overview of our methodology and explained how we created our datasets. We downloaded applications from F-Droid and obfuscated the applications with identifier renaming, arithmetic branch insertion, call indirection, goto insertion, method overload, nop insertion, reflection, reorder, constant string encryption methods using obfuscapk. We also provided brief descriptions of the applied obfuscation techniques. For the disassembling process, we used a popular disassembler apktool which is also integrated with obfuscapk. Then we continued our discussion with the forms of our datasets, namely ISD and MSD. We employed NLP techniques to generate language models from these datasets and utilized LSTM architecture. Hence, we provided details on how we implemented the LSTM model. First, we explained the imported modules and libraries we used to build the LSTM architecture, which are Tensorflow, Keras, Pickles, Scikit-learn, Numpy, Matplotlib, and Seaborn. Then we described the pipeline we used for training and testing the model. Lastly, we provided the configuration details and selection of the parameter values for the LSTM model.

CHAPTER 4

RESULTS

4.1 Experimental Results

In this section, we present the results of each obfuscation detection model for both instruction sequence (ISD) and method sequence (MSD) datasets. For the ISD and the MSD, the number of correctly and incorrectly classified samples for each detection unit are shown in confusion matrices that can be found in Appendix A. A reference confusion matrix is also given in Figure 4.1.

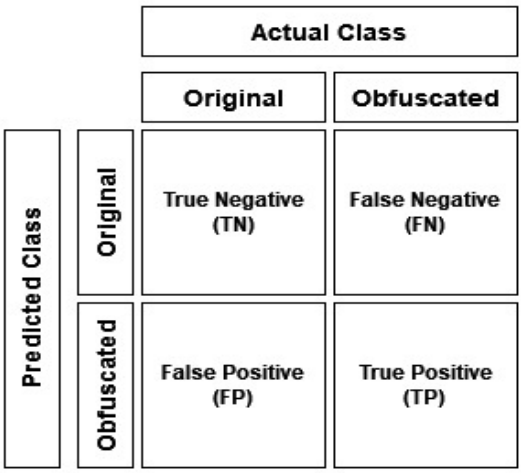


Figure 4.1: Confusion matrix for reference

The value true positive (TP) in the confusion matrix refers to the correctly recognized obfuscated instructions in ISD and correctly recognized obfuscated methods in MSD, whereas true negative (TN) refers to the correctly recognized non-obfuscated instructions or methods. On the other hand, false positive (FP) refers to the non-obfuscated instructions or methods identified as obfuscated. Lastly, false negative (FN) refers to the obfuscated instructions or methods recognized as non-obfuscated. From these values, we calculated the true positive rate (TPR), false positive rate (FPR), and accuracy rate (ACC) of the models. TPR refers to the ratio of the correctly identified positive predictions in all positive cases and is calculated as follows:

$$TPR = \frac{TP}{(TP + FN)} \tag{4.1}$$

FPR states the proportion of incorrectly identified positive predictions in all negative cases and calculated as follows:

$$FPR = \frac{FP}{(FP + TN)} \tag{4.2}$$

Lastly, ACC is the measure of the proportion of correct predictions in our model out of all predictions and calculated as follows:

$$ACC = \frac{(TP + TN)}{(TP + FP + TN + FN)} \tag{4.3}$$

TPR, FPR, and ACC values of the models can be seen on Table 4.1.

Table 4.1: The results obtained from the models trained by the Instruction Sequence Dataset (ISD) and the Method Sequence Dataset (MSD)

	ISD			MSD		
	TPR	FPR	ACC	TPR	FPR	ACC
Identifier Renaming	99.86%	42.66%	62.79%	99.19%	10.21%	94.00%
Arithmetic Branch Insertion	50.02%	0.11%	54.41%	98.99%	39.04%	67.94%
Call Indirection	50.68%	27.75%	54.26%	88.07%	43.79%	71.92%
Goto Insertion	100.00%	50.00%	54.77%	99.99%	1.29%	99.35%
Method Overload	82.66%	49.99%	52.49%	49.96%	0.00%	56.97%
Nop Insertion	98.85%	33.14%	79.26%	98.69%	0.01%	99.33%
Reflection	59.96%	48.80%	59.47%	51.78%	45.69%	52.52%
Reorder	99.34%	29.45%	82.62%	98.71%	0.01%	99.34%
Constant String Encryption	50.61%	34.15%	52.41%	54.32%	5.82%	60.33%

Obfuscation detection accuracy ranges between 52.41% and 82.62% in the ISD model. Method overload and constant string encryption techniques reveal relatively low accuracy rates, 52.49%, and 52.41%, respectively. Arithmetic branch insertion, call indirection, and goto insertion methods have similarly low accuracy values, 54.41%, 54.26%, and 54.77% in the ISD model. A slight increase is observed in reflection and identifier renaming detection. The highest accuracies of the ISD model belong to nop insertion and reorder methods (79.26% and 82.62%). Also, in the ISD models, reflection detection has the highest TPR, whereas arithmetic branch insertion has the highest FPR. When we switch to the MSD model, the accuracies of some methods increase drastically. Reorder, nop insertion, and goto insertion methods acquire high accuracy rates (99.34%, 99.33%, and 99.35%, respectively). Identifier renaming also shows a significant increase (94.0% accuracy). Except for the reflection method, all the techniques returned higher accuracy values in the MSD model than in the ISD model.

4.2 Discussion

In the present study, we investigated the performance of the LSTM model, which was designed by two application representations for detecting nine obfuscation methods, namely, identifier renaming, arithmetic branch insertion, call indirection, goto

insertion, method overload, nop insertion, reflection, reorder, and constant string encryption. We applied NLP techniques and examined if it is feasible to model and treat original and obfuscated Smali sequences as different languages for obfuscation detection. While using the ISD, the model achieves the TPR of 99.86%, 100.00%, 98.85%, and 99.43% for identifier renaming, goto insertion, nop insertion, and reorder, respectively. However, due to the high FPR values, the model cannot perform accurate detections using the ISD. The accuracy values vary between 52.41% and 82.62% while using the ISD representation. Another interesting result is that arithmetic branch insertion achieves 0.11% FPR in the ISD, but since the TPR of arithmetic branch insertion is 50.02%, the model could only reach the accuracy value of 54.41%. The results showed that the ISD modeling approach performed relatively low, compared to the MSD modeling, in detecting the majority of the obfuscation methods. This outcome may be due to the large number of common instruction sequences used in both original and obfuscated samples. Therefore, according to the experimental results using instruction as the basic unit while modeling smali language to detect obfuscation does not seem practical. Nevertheless, the ISD model achieved promising accuracy in reorder detection (82.62%) and nop insertion detection (79.26%). On the other hand, the MSD modeling approach achieved great accuracy values in detecting most obfuscation techniques.

When we switched to the MSD, we observed the high TPR values of 99.19%, 98.99%, 99.99%, 98.69%, and 98.71% for identifier renaming, arithmetic branch insertion, goto insertion, nop insertion, and reorder, respectively. Also, except for arithmetic branch insertion and call indirection, all obfuscation techniques achieved lower FPR values than the ISD. The lowest FPR values of the MSD are 0.00% for method overload, 0.01% for nop insertion, and 0.01% for reorder. A peculiar characteristic of MSD modeling is that contextual information and functionality increase as one proceeds from instructions to methods, leading to efficient learning and higher accuracy values. In particular, reorder, nop insertion, and goto insertion (cf. control flow obfuscations) reached accuracies around 99%. An exception is that the accuracy of reflection detection was lower in the MSD model than in the ISD model. In the MSD approach, the model also reached a high accuracy of 94% for identifier renaming detection. Detecting identifier renaming is problematic since numerous renaming schemes are used in the wild. Specifically, our MSD model was good at detecting the renaming pattern of obfuscapk. Finally, the other control flow obfuscation techniques, namely arithmetic branch insertion and call indirection, achieved an accuracy of 67,94% and 71,92%, respectively, pointing to a need for further improvement. Overall, the results suggest that NLP techniques and LSTM are methodologically suitable for detecting most control flow obfuscations.

Let us continue our discussion by dissecting the interesting experimental results for several obfuscation techniques. In identifier renaming detection, the model achieved similar TPR rates in both the ISD and the MSD. Nonetheless, the FPR value decreases significantly from 42.66% to 10.21% when we use the MSD instead of the ISD. This decrease in the FPR of identifier renaming results in the accuracy value of 94.00%, which is the highest achieved accuracy value after control flow obfuscations. In detecting arithmetic branch insertion, even though the model achieved the lowest FPR in the ISD, it achieved a low accuracy value of 54.41%. When we use the MSD for detecting arithmetic branch insertion, the TPR value increases to 98.99% from

50.02%. However, the FPR also increases drastically from 0.11% to 39.04%. Therefore, the model cannot identify arithmetic branch insertion successfully, although it is another control flow obfuscation technique. The model also performs a similar behavior for call indirection detection. The TPR and FPR values increase when we switch from the ISD to the MSD. Since the increase in the FPR is relatively lower than the arithmetic branch insertion, the model achieves a slightly better accuracy value for call indirection detection than arithmetic branch insertion detection. Then in the goto insertion detection, the model obtained similar high TPR rates for both representations. Similar to identifier renaming, the FPR drops to 1.29% from 50.00% when we use the MSD in goto insertion detection. We mentioned that the contextual information, semantics, and functionality increase as we go from instructions to methods similar to natural languages. Therefore, the decreases in the FPR values are expected when we switch to the MSD. Moreover, in method overload detection, the FPR decreases significantly from 49.99% to 0.00% in the MSD. Nevertheless, the TPR also decreases from 82.66% to 49.96%. Therefore, although the accuracy value increases in the MSD, it is not as much as we expected. Lastly, we should mention that our model performed best for reorder detection in both the ISD and the MSD.

We also mentioned in the Dataset section that some obfuscation methods we applied include other obfuscation techniques. For example, reorder technique employs both nop insertion and goto insertion while transforming the source code. Moreover, arithmetic branch insertion also inserts goto instructions after adding random arithmetic operations. Therefore, we expect to see correlations between the detection accuracies of these techniques. Our model achieved accuracy values around 99% for nop insertion and goto insertion separately. In the reorder detection case, we can see the expected correlation since reorder detection achieved an accuracy value of 99.34% while using the MSD. Reorder obfuscation utilizes nop and goto instructions heavily. Therefore, our LSTM model could easily learn the patterns of reorder obfuscation that includes nop and goto instructions and achieved similar and high accuracy values as the detection of nop insertion and goto insertion. On the other hand, we also expect to see a correlation between the detection accuracies of arithmetic branch insertion and goto insertion. Nonetheless, unlike in the reorder detection case, we did not observe the expected correlation for arithmetic branch insertion. Our model achieved a detection accuracy of 67.94% for arithmetic branch insertion. In the present study, we used a black-box model that limits the explainability of our model since we do not know the complete inner working of the neural network architectures. However, we suspect that the low ratio of added goto instructions to the other additions to the source code in the arithmetic branch insertion method may have caused this inconsistency. Due to the aforementioned low ratio, the pattern of goto insertion may become ambiguous for the model to grasp. Regardless, a further investigation is needed to explain the low accuracy value of arithmetic branch insertion since it has the lowest detection accuracy among the investigated control flow obfuscation methods in the present study.

A comparative analysis of the results is challenging since different tools use different algorithms and patterns that will reflect differences in the code even if the same obfuscation methods are used. For instance, [6] and [7] employ signature-based approaches, whereas [107], [108], and [109] utilize ML-based methods by using different tools for implementing obfuscation. Moreover, some previous work investigates

the control flow obfuscation techniques wholistically. Therefore, in this section, we present a partial discussion of the findings within the literature framework of the previous research.

Another significant challenge in model training is that malware samples are usually obfuscated. Including obfuscated malware in the dataset may lead to errors while training the model. To overcome this problem, researchers typically build their own dataset, for instance, by collecting apks from F-Droid and obfuscating them using different tools [109]. In the present study, we built a dataset more extensive than the ones available in the literature.

We studied specific control flow obfuscations in the present study and reported the results separately for reorder, nop insertion, goto insertion, arithmetic branch insertion, call indirection, and reflection. In [108], researchers report an accuracy of 91% for identifier renaming, 80% for string encryption, and 66% for control flow obfuscation detection. They included both benign and malware samples in the dataset. Our model obtained better accuracy results for identifier renaming. Nevertheless, we cannot compare the accuracy values fully since we tested specific control flow obfuscation methods. In [109], the detection accuracies of 70%, 77%, 80%, and 79% were reported for identifier renaming, string encryption, control flow obfuscation, and reflection, respectively. Also, in a similar study [110] that focused on detecting the same obfuscation methods as in [108], researchers employed GCN (Graph Convolutional Networks) and LSTM together with supervised learning and achieved an accuracy of around 98%. Our model has the advantage of reaching better or similar accuracy results for identifier renaming and control flow obfuscations than [108], [109], and [110]. Also, these studies need feature extraction process that requires expertise and a longer processing time. Our approach employs direct classification, which requires relatively little effort than previous studies. Another study, which investigated control flow obfuscation methods separately, utilized a direct classification approach with LTSM without separating junk code insertion methods [64]. Our LSTM model outperforms the LSTM model in [64] in detecting all the applied obfuscation methods. The main difference between this study and ours is that they take opcodes as basic units, whereas we take opcodes and operands together as instructions. Moreover, we created two datasets for two different data representations of applications to be forwarded to our LSTM model, whereas the study [64] trained their model only with opcode sequences.

CHAPTER 5

CONCLUSION

5.1 Conclusion

The prevalence and acceptance of the Android operating system on mobile devices have led to the development of many benign and malicious applications for Android. Android applications are relatively easy to disassemble and decompile, making them vulnerable to reverse engineering. Therefore, software developers and malware authors frequently use obfuscation techniques against reverse engineering. Software developers employ obfuscation to protect their intellectual property and economic gain. On the other hand, malware developers use it to evade detection engines. Therefore, it is crucial to detect which obfuscation techniques are employed in an Android application to build more efficient systems in various research areas such as digital forensics, clone detection, third-party library detection, and malware detection.

Obfuscation detection methodologies follow similar trends as other detection methodologies, especially malware detection. Previous studies in Android obfuscation detection mainly used signature-based and ML-based approaches. Signature-based obfuscation detection methodologies have some limitations. First of all, signature generation is a laborious process, and generated signatures are primarily based on specific configurations of popular obfuscation tools. Therefore, they are ineffective against new obfuscation techniques and tools. On the other hand, ML-based approaches performed better than signature-based methodologies. However, they still need feature extraction, which is an effortful process that also requires the researcher's expertise. Also, they are insufficient to detect new types of obfuscation techniques. Therefore, similar to malware detection, deep learning-based approaches have gained importance in obfuscation detection. DL methods can automatically grasp the semantics and patterns to derive the meaning of new data without feature extraction and expert knowledge. Thus, providing more reliable and automated systems for detection tasks.

This thesis presents an approach that utilizes NLP techniques and the LSTM model. We collected apks and obfuscated them with nine popular techniques using obfuscapk. Then we disassembled original and obfuscated samples with apktool statically to obtain the smali codes of the applications. We treated the smali codes of the applications as if they were natural languages due to the similarities between natural languages and programming languages. We chose the basic unit of smali code as instructions which constitute opcodes and operands. Then we built our datasets, the

ISD and the MSD, in which we represented applications as instruction sequences and method sequences. We created the ISD and the MSD for each obfuscation method separately.

In this study, we also wanted to examine the feasibility of representing original and obfuscated samples as different languages and the performance of the LSTM model in detecting different obfuscation techniques. Therefore, we trained nine LSTM models for obfuscation techniques separately using the generated datasets. We experimented with several parameters while training the LSTM model to obtain the optimal values for our study. Our system obtained the highest detection accuracies for reorder, nop insertion, goto insertion, and identifier renaming; 99.34%, 99.33%, 99.35%, and 94.00%, respectively, while using the MSD. Additionally, the false positive rates for reorder, nop insertion, goto insertion, and identifier renaming is 0.01%, 0.01%, 1.29%, 10.21%, respectively. We observed that representing applications as method sequences is better for detecting obfuscation and achieved higher accuracy than representing applications as instructions. Since methods contain more semantic information than instructions, this outcome was expected. Moreover, we can conclude that our model is a suitable candidate for control flow obfuscation detection as it achieves the highest accuracies according to the experimental results. Also, our model does not require any feature extraction and time-consuming preprocessing stages, making it a more practical model.

5.2 Limitations and Future Work

We mentioned that there are various obfuscation methods, and it is nearly impossible to create a dataset covering all of them. Due to its ease of use, we utilized obfuscapk to obfuscate the collected applications. Even though we did not use feature extraction, the learned patterns by the model may be specific to the obfuscation implementation logic of obfuscapk. This singularity may be a limitation against the generalizability of our model. Therefore, we will enrich the dataset by using different obfuscation tools as future work. Similarly, the output of the disassembler tool may cause different accuracy values due to different code representations. We plan to use other disassemblers and train the models again to see if there will be any difference in the accuracy values.

Besides that, we will experiment with different RNNs and Transformer architectures to examine if we can increase the detection accuracy of the methods with low rates, such as string encryption, method overloading, call indirection, and reflection. However, the mentioned neural network architectures, including LSTM, are considered black-box models. In black-box models, developers do not have control over the model's inner workings, and even the designers of the models cannot explain why a neural network model made a particular conclusion. This situation limits the explainability of the studies that apply DL methods, including ours. Since we cannot explain the internals of the models, our model's accuracy may be closely related to our dataset, which is a limitation against the generalizability of our model. This problem is common in DL-based detection studies. Therefore, the research community

focused on explainable deep learning models recently. In this context, we will also look into explainable deep learning models for obfuscation detection as another future work.

Lastly, applications are obfuscated with multiple techniques in the wild rather than just one. Therefore, we will start investigating the detection of multilevel obfuscation with multiclass classification. We will also collect real-world samples and investigate the latest obfuscation trends in the Android environment in future work.

REFERENCES

[1] [Online]. Available: https://colah.github.io/posts/2015-08-Understanding-LSTMs/, Accessed on: Feb 2022.

[2] [Online]. Available: https://gs.statcounter.com/os-market-share/mobile/worldwide, Accessed on: Feb 2022.

[3] [Online]. Available: https://www.appbrain.com/stats/number-of-android-apps, Accessed on: Feb 2022.

[4] Kaspersky. [Online]. Available: https://securelist.com/it-threat-evolution-q2-2021-mobile-statistics/103636/, Accessed on: Feb 2022.

[5] G. Hecht, C. Neverov, and A. Bergel, "Vision: Alleviating Android developer burden on obfuscation," in *Proceedings of the IEEE/ACM 7th International Conference on Mobile Software Engineering and Systems*, pp. 137–141, 2020.

[6] D. Wermke, N. Huaman, Y. Acar, B. Reaves, P. Traynor, and S. Fahl, "A large scale investigation of obfuscation use in Google Play," in *Proceedings of the 34th Annual Computer Security Applications Conference*, pp. 222–235, 2018.

[7] S. Dong, M. Li, W. Diao, X. Liu, J. Liu, Z. Li, F. Xu, K. Chen, X. Wang, and K. Zhang, "Understanding Android obfuscation techniques: A large-scale investigation in the wild," in *International Conference on Security and Privacy in Communication Systems*, pp. 172–192, Springer, 2018.

[8] Y. Peng, Y. Chen, and B. Shen, "An Adaptive Approach to Recommending Obfuscation Rules for Java Bytecode Obfuscators," in *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, vol. 1, pp. 97–106, IEEE, 2019.

[9] H. Xu, Y. Zhou, J. Ming, and M. Lyu, "Layered obfuscation: A taxonomy of software obfuscation techniques for layered security," *Cybersecurity*, vol. 3, no. 1, pp. 1–18, 2020.

[10] A. Sheneamer, S. Roy, and J. Kalita, "A detection framework for semantic code clones and obfuscated code," *Expert Systems with Applications*, vol. 97, pp. 405–420, 2018.

[11] B. Kim, K. Lim, S.-J. Cho, and M. Park, "Romadroid: A robust and efficient technique for detecting Android app clones using a tree structure and components of each app's manifest file," *IEEE Access*, vol. 7, pp. 72182–72196, 2019.

[12] J. Guo, D. Liu, R. Zhao, and Z. Li, "WLTDroid: Repackaging Detection Approach for Android Applications," in *International Conference on Web Information Systems and Applications*, pp. 579–591, Springer, 2020.

[13] Z. He, G. Ye, L. Yuan, Z. Tang, X. Wang, J. Ren, W. Wang, J. Yang, D. Fang, and Z. Wang, "Exploiting binary-level code virtualization to protect Android applications against app repackaging," *IEEE Access*, vol. 7, pp. 115062–115074, 2019.

[14] M. Hammad, J. Garcia, and S. 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*, pp. 421–431, 2018.

[15] C. Sun, H. Zhang, S. Qin, J. Qin, Y. Shi, and Q. Wen, "DroidPDF: The obfuscation resilient packer detection framework for Android apps," *IEEE Access*, vol. 8, pp. 167460–167474, 2020.

[16] S. Millar, N. McLaughlin, J. Martinez del Rincon, P. Miller, and Z. Zhao, "Dandroid: A multi-view discriminative adversarial network for obfuscated Android malware detection," in *Proceedings of the Tenth ACM Conference on Data and Application Security and Privacy*, pp. 353–364, 2020.

[17] W. Y. Lee, J. Saxe, and R. Harang, "SeqDroid: Obfuscated Android malware detection using stacked convolutional and recurrent neural networks," in *Deep Learning Applications for Cyber Security*, pp. 197–210, Springer, 2019.

[18] X. Zhan, L. Fan, T. Liu, S. Chen, L. Li, H. Wang, Y. Xu, X. Luo, and Y. Liu, "Automated third-party library detection for Android applications: Are we there yet?," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 919–930, IEEE, 2020.

[19] Y. Wang, H. Wu, H. Zhang, and A. Rountev, "Orlis: Obfuscation-resilient library detection for Android," in *2018 IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pp. 13–23, IEEE, 2018.

[20] L. Deshotels, V. Notani, and A. Lakhotia, "Droidlegacy: Automated familial classification of Android malware," in *Proceedings of ACM SIGPLAN on Program Protection and Reverse Engineering Workshop 2014*, pp. 1–12, 2014.

[21] M. Zheng, M. Sun, and J. C. Lui, "Droid analytics: A signature based analytic system to collect, extract, analyze and associate Android malware," in *2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, pp. 163–171, IEEE, 2013.

[22] J. Li, L. Sun, Q. Yan, Z. Li, W. Srisa-An, and H. Ye, "Significant permission identification for machine-learning-based Android malware detection," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 7, pp. 3216–3225, 2018.

[23] S. Chen, M. Xue, Z. Tang, L. Xu, and H. Zhu, "Stormdroid: A streaminglized machine learning-based system for detecting Android malware," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pp. 377–388, 2016.

[24] R. Vinayakumar, K. Soman, and P. Poornachandran, "Deep Android malware detection and classification," in *2017 International Conference on Advances*

in Computing, Communications and Informatics (ICACCI), pp. 1677–1683, IEEE, 2017.

[25] X. Xiao, S. Zhang, F. Mercaldo, G. Hu, and A. K. Sangaiah, "Android malware detection based on system call sequences and LSTM," *Multimedia Tools and Applications*, vol. 78, no. 4, pp. 3979–3999, 2019.

[26] C. Acarturk, M. Sirlanci, P. G. Balikcioglu, D. Demirci, N. Sahin, and O. A. Kucuk, "Malicious Code Detection: Run Trace Output Analysis by LSTM," *IEEE Access*, vol. 9, pp. 9625–9635, 2021.

[27] M. Sundermeyer, R. Schlüter, and H. Ney, "LSTM neural networks for language modeling," in *Thirteenth Annual Conference of the International Speech Communication Association*, 2012.

[28] M. Sundermeyer, H. Ney, and R. Schlüter, "From feedforward to recurrent LSTM neural networks for language modeling," *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, vol. 23, no. 3, pp. 517–529, 2015.

[29] [Online]. Available: https://developer.android.com/guide/components/fundamentals, Accessed on: Feb 2022.

[30] D. Maiorca, D. Ariu, I. Corona, M. Aresu, and G. Giacinto, "Stealth attacks: An extended insight into the obfuscation effects on Android malware," *Computers & Security*, vol. 51, pp. 16–31, 2015.

[31] Apktool. [Online]. Available: https://ibotpeaches.github.io/Apktool/, Accessed on: Jan 2022.

[32] JEB. [Online]. Available: https://www.pnfsoftware.com/jeb/, Accessed on: Feb 2022.

[33] Baksmali. [Online]. Available: https://github.com/JesusFreke/smali, Accessed on: Feb 2022.

[34] IDA Pro. [Online]. Available: https://hex-rays.com/ida-pro/, Accessed on: Feb 2022.

[35] Androguard. [Online]. Available: https://github.com/androguard/androguard, Accessed on: Feb 2022.

[36] dex2jar. [Online]. Available: https://github.com/pxb1988/dex2jar, Accessed on: Feb 2022.

[37] H. Xu, Y. Zhou, Y. Kang, and M. R. Lyu, "On secure and usable program obfuscation: A survey," *arXiv preprint arXiv:1710.01139*, 2017.

[38] S. Garg, C. Gentry, S. Halevi, M. Raykova, A. Sahai, and B. Waters, "Candidate indistinguishability obfuscation and functional encryption for all circuits," *SIAM Journal on Computing*, vol. 45, no. 3, pp. 882–929, 2016.

[39] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," 1997.

[40] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang, "On the (im) possibility of obfuscating programs," in *Annual International Cryptology Conference*, pp. 1–18, Springer, 2001.

[41] S. Goldwasser and Y. T. Kalai, "On the impossibility of obfuscation with auxiliary input," in *46th Annual IEEE Symposium on Foundations of Computer Science (FOCS'05)*, pp. 553–562, IEEE, 2005.

[42] S. Schrittwieser, S. Katzenbeisser, J. Kinder, G. Merzdovnik, and E. Weippl, "Protecting software through obfuscation: Can it keep pace with progress in code analysis?," *ACM Computing Surveys (CSUR)*, vol. 49, no. 1, pp. 1–37, 2016.

[43] M. Dalla Preda and R. Giacobazzi, "Semantics-based code obfuscation by abstract interpretation," *Journal of Computer Security*, vol. 17, no. 6, pp. 855–908, 2009.

[44] T. László and Á. Kiss, "Obfuscating C++ programs via control flow flattening," *Annales Universitatis Scientarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica*, vol. 30, no. 1, pp. 3–19, 2009.

[45] W. Wang, M. Li, Z. Tang, H. Wang, G. Ye, F. Wang, J. Ren, X. Gong, D. Fang, and Z. Wang, "Invalidating analysis knowledge for code virtualization protection through partition diversity," *IEEE Access*, vol. 7, pp. 169160–169173, 2019.

[46] S. Banescu and A. Pretschner, "A tutorial on software obfuscation," *Advances in Computers*, vol. 108, pp. 283–353, 2018.

[47] N. Viennot, E. Garcia, and J. Nieh, "A measurement study of Google Play," in *The 2014 ACM International Conference on Measurement and Modeling of Computer Systems*, pp. 221–233, 2014.

[48] S. Luo and P. Yan, "Fake apps feigning legitimacy," *A Trend Micro Research Paper*, 2014.

[49] [Online]. Available: https://developer.android.com/studio/build/shrink-code, Accessed on: Feb 2022.

[50] V. Sihag, M. Vardhan, and P. Singh, "A survey of Android application and malware hardening," *Computer Science Review*, vol. 39, p. 100365, 2021.

[51] M. D. Preda and F. Maggi, "Testing Android malware detectors against code obfuscation: A systematization of knowledge and unified methodology," *Journal of Computer Virology and Hacking Techniques*, vol. 13, no. 3, pp. 209–232, 2017.

[52] Y. Zhou and X. Jiang, "Dissecting Android malware: Characterization and evolution," in *2012 IEEE Symposium on Security and Privacy*, pp. 95–109, IEEE, 2012.

[53] J. Crussell, C. Gibler, and H. Chen, "Andarwin: Scalable detection of semantically similar Android applications," in *European Symposium on Research in Computer Security*, pp. 182–199, Springer, 2013.

[54] Trend Micro. [Online]. Available: https://blog.trendmicro.com/trendlabs-security-intelligence/a-look-into-repackaged-apps-and-its-role-in-the-mobile-threat-landscape/, Accessed on: Jan 2022.

[55] A. Merlo, A. Ruggia, L. Sciolla, and L. Verderame, "You shall not repackage! demystifying anti-repackaging on Android," *Computers & Security*, vol. 103, p. 102181, 2021.

[56] V. Rastogi, Y. Chen, and X. Jiang, "Catch me if you can: Evaluating Android anti-malware against transformation attacks," *IEEE Transactions on Information Forensics and Security*, vol. 9, no. 1, pp. 99–108, 2013.

[57] P. Kanani, K. Srivastava, J. Gandhi, D. Parekh, and M. Gala, "Obfuscation: Maze of code," in *2017 2nd International Conference on Communication Systems, Computing and IT Applications (CSCITA)*, pp. 11–16, IEEE, 2017.

[58] P. Schulz, "Code protection in Android," *Insititute of Computer Science, Rheinische Friedrich-Wilhelms-Universitgt Bonn, Germany*, vol. 110, 2012.

[59] Y. Zhou, A. Main, Y. X. Gu, and H. Johnson, "Information hiding in software with mixed boolean-arithmetic transforms," in *International Workshop on Information Security Applications*, pp. 61–75, Springer, 2007.

[60] P. Faruki, H. Fereidooni, V. Laxmi, M. Conti, and M. Gaur, "Android code protection via obfuscation techniques: Past, present and future directions," *arXiv preprint arXiv:1611.10231*, 2016.

[61] C. Collberg, C. Thomborson, and D. Low, "Manufacturing cheap, resilient, and stealthy opaque constructs," in *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 184–196, 1998.

[62] J. Palsberg, S. Krishnaswamy, M. Kwon, D. Ma, Q. Shao, and Y. Zhang, "Experience with software watermarking," in *Proceedings 16th Annual Computer Security Applications Conference (ACSAC'00)*, pp. 308–316, IEEE, 2000.

[63] A. Majumdar and C. Thomborson, "Manufacturing opaque predicates in distributed systems for code obfuscation," in *Proceedings of the 29th Australasian Computer Science Conference-Volume 48*, pp. 187–196, Citeseer, 2006.

[64] A. Bacci, A. Bartoli, F. Martinelli, E. Medvet, and F. Mercaldo, "Detection of obfuscation techniques in Android applications," in *Proceedings of the 13th International Conference on Availability, Reliability and Security*, pp. 1–9, 2018.

[65] V. Rastogi, Y. Chen, and X. Jiang, "Droidchameleon: Evaluating Android anti-malware against transformation attacks," in *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, pp. 329–334, 2013.

[66] A. Apvrille and R. Nigam, "Obfuscation in Android malware, and how to fight back," *Virus Bulletin*, pp. 1–10, 2014.

[67] Allatori. [Online]. Available: http://www.allatori.com/, Accessed on: Jan 2022.

[68] Dasho. [Online]. Available: https://www.preemptive.com/products/dasho/, Accessed on: Jan 2022.

[69] J. Park, H. Kim, Y. Jeong, S.-j. Cho, S. Han, and M. Park, "Effects of Code Obfuscation on Android App Similarity Analysis," *J. Wirel. Mob. Networks Ubiquitous Comput. Dependable Appl.*, vol. 6, no. 4, pp. 86–98, 2015.

[70] Dexprotector. [Online]. Available: https://dexprotector.com/, Accessed on: Jan 2022.

[71] H. Cho, J. H. Yi, and G.-J. Ahn, "Dexmonitor: Dynamically analyzing and monitoring obfuscated Android applications," *IEEE Access*, vol. 6, pp. 71229–71240, 2018.

[72] DexGuard. [Online]. Available: https://www.guardsquare.com/dexguard, Accessed on: Jan 2022.

[73] H. Gonzalez, A. A. Kadir, N. Stakhanova, A. J. Alzahrani, and A. A. Ghorbani, "Exploring reverse engineering symptoms in Android apps," in *Proceedings of the Eighth European Workshop on System Security*, pp. 1–7, 2015.

[74] Proguard. [Online]. Available: https://www.guardsquare.com/en/products/proguard, Accessed on: Jan 2022.

[75] M. Zheng, P. P. Lee, and J. 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*, pp. 82–101, Springer, 2012.

[76] AAMO. [Online]. Available: https://github.com/necst/aamo/, Accessed on: Jan 2022.

[77] 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.

[78] P. Faruki, V. Ganmoor, V. Laxmi, M. S. Gaur, and A. Bharmal, "AndroSimilar: Robust statistical feature signature for Android malware detection," in *Proceedings of the 6th International Conference on Security of Information and Networks*, pp. 152–159, 2013.

[79] J. Chen, M. H. Alalfi, T. R. Dean, and Y. Zou, "Detecting Android malware using clone detection," *Journal of Computer Science and Technology*, vol. 30, no. 5, pp. 942–956, 2015.

[80] S. Alam, R. Riley, I. Sogukpinar, and N. Carkaci, "Droidclone: Detecting Android malware variants by exposing code clones," in *2016 Sixth International Conference on Digital Information and Communication Technology and its Applications (DICTAP)*, pp. 79–84, IEEE, 2016.

[81] J. Zhang, A. R. Beresford, and S. A. Kollmann, "Libid: Reliable identification of obfuscated third-party Android libraries," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 55–65, 2019.

[82] J. Senanayake, H. Kalutarage, and M. O. Al-Kadri, "Android Mobile Malware Detection Using Machine Learning: A Systematic Review," *Electronics*, vol. 10, no. 13, p. 1606, 2021.

[83] Q. Wu, X. Zhu, and B. Liu, "A survey of Android malware static detection technology based on machine learning," *Mobile Information Systems*, vol. 2021, 2021.

[84] G. Baldini and D. Geneiatakis, "A performance evaluation on distance measures in kNN for mobile malware detection," in *2019 6th International Conference on Control, Decision and Information Technologies (CoDIT)*, pp. 193–198, IEEE, 2019.

[85] B. Sanz, I. Santos, C. Laorden, X. Ugarte-Pedrero, J. Nieves, P. G. Bringas, and G. Álvarez Marañón, "MAMA: Manifest analysis for malware detection in Android," *Cybernetics and Systems*, vol. 44, no. 6-7, pp. 469–488, 2013.

[86] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens, "Drebin: Effective and explainable detection of Android malware in your pocket.," in *NDSS*, vol. 14, pp. 23–26, 2014.

[87] N. Milosevic, A. Dehghantanha, and K.-K. R. Choo, "Machine learning aided Android malware classification," *Computers & Electrical Engineering*, vol. 61, pp. 266–274, 2017.

[88] K. Liu, S. Xu, G. Xu, M. Zhang, D. Sun, and H. Liu, "A review of Android malware detection approaches based on machine learning," *IEEE Access*, vol. 8, pp. 124579–124607, 2020.

[89] A. Narayanan, L. Chen, and C. K. Chan, "Addetect: Automated detection of Android ad libraries using semantic analysis," in *2014 IEEE Ninth International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP)*, pp. 1–6, IEEE, 2014.

[90] B. Liu, B. Liu, H. Jin, and R. Govindan, "Efficient privilege de-escalation for ad libraries in mobile apps," in *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, pp. 89–103, 2015.

[91] Q. U. Ain, W. H. Butt, M. W. Anwar, F. Azam, and B. Maqbool, "A systematic review on code clone detection," *IEEE Access*, vol. 7, pp. 86121–86144, 2019.

[92] [Online]. Available: https://en.wikipedia.org/wiki/Backpropagation, Accessed on: Feb 2022.

[93] [Online]. Available: https://en.wikipedia.org/wiki/Types_of_artificial_neural_networks, Accessed on: Feb 2022.

[94] IBM. [Online]. Available: https://www.ibm.com/cloud/learn/recurrent-neural-networks, Accessed on: Feb 2022.

[95] [Online]. Available: https://towardsdatascience.com/lstm-networks-a-detailed-explanation-8fae6aefc7f9, Accessed on: Feb 2022.

[96] IBM. [Online]. Available: https://www.ibm.com/cloud/learn/natural-language-processing, Accessed on: Feb 2022.

[97] [Online]. Available: https://machinelearningmastery.com/natural-language-processing/ , Accessed on: Feb 2022.

[98] D. W. Otter, J. R. Medina, and J. K. Kalita, "A survey of the usages of deep learning for natural language processing," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 32, no. 2, pp. 604–624, 2020.

[99] A. Torfi, R. A. Shirvani, Y. Keneshloo, N. Tavaf, and E. A. Fox, "Natural language processing advancements by deep learning: A survey," *arXiv preprint arXiv:2003.01200*, 2020.

[100] S. Sen and B. Can, "Android Security using NLP Techniques: A Review," *arXiv preprint arXiv:2107.03072*, 2021.

[101] J. Qiu, J. Zhang, W. Luo, L. Pan, S. Nepal, and Y. Xiang, "A survey of Android malware detection with deep neural models," *ACM Computing Surveys (CSUR)*, vol. 53, no. 6, pp. 1–36, 2020.

[102] E. B. Karbab, M. Debbabi, A. Derhab, and D. Mouheb, "MalDozer: Automatic framework for Android malware detection using deep learning," *Digital Investigation*, vol. 24, pp. S48–S59, 2018.

[103] Z. Wu, X. Chen, and S. U.-J. Lee, "Identifying Latent Android Malware from Application's Description using LSTM," in *Proceedings of International Conference on Information, System and Convergence Applications*, pp. 40–42, 2019.

[104] K. Xu, Y. Li, R. H. Deng, and K. Chen, "Deeprefiner: Multi-layer Android malware detection system applying deep neural networks," in *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pp. 473–487, IEEE, 2018.

[105] M. Amin, T. A. Tanveer, M. Tehseen, M. Khan, F. A. Khan, and S. Anwar, "Static malware detection and attribution in Android byte-code through an end-to-end deep system," *Future Generation Computer Systems*, vol. 102, pp. 112–126, 2020.

[106] A. Hota and P. Irolla, "Deep Neural Networks for Android Malware Detection," in *ICISSP*, pp. 657–663, 2019.

[107] 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.

[108] O. Mirzaei, J. M. de Fuentes, J. Tapiador, and L. Gonzalez-Manzano, "AndrODet: An adaptive Android obfuscation detector," *Future Generation Computer Systems*, vol. 90, pp. 240–261, 2019.

[109] M. Park, G. You, S.-j. 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.

[110] 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.

[111] Y. Zhang, G. Xiao, Z. Zheng, T. Zhu, I. W. Tsang, and Y. Sui, "An Empirical Study of Code Deobfuscations on Detecting Obfuscated Android Piggybacked Apps," in *2020 27th Asia-Pacific Software Engineering Conference (APSEC)*, pp. 41–50, IEEE, 2020.

[112] W. Yoo, M. Ji, M. Kang, and J. H. Yi, "String deobfuscation scheme based on dynamic code extraction for mobile malwares," *IT Convergence Practice*, vol. 4, no. 2, pp. 1–8, 2016.

[113] B. Bichsel, V. Raychev, P. Tsankov, and M. Vechev, "Statistical deobfuscation of Android applications," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pp. 343–355, 2016.

[114] J. He, P. Ivanov, P. Tsankov, V. Raychev, and M. Vechev, "Debin: Predicting debug information in stripped binaries," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1667–1680, 2018.

[115] Z. Kan, H. Wang, L. Wu, Y. Guo, and D. X. Luo, "Automated deobfuscation of Android native binary code," *arXiv preprint arXiv:1907.06828*, 2019.

[116] Y. Zhao, Z. Tang, G. Ye, D. Peng, D. Fang, X. Chen, and Z. Wang, "Semantics-aware obfuscation scheme prediction for binary," *Computers & Security*, vol. 99, p. 102072, 2020.

[117] F-Droid. [Online]. Available: https://f-droid.org/, Accessed on: Jan 2022.

[118] TRUBA Wiki. [Online]. Available: http://wiki.grid.org.tr/index.php/TRUBA-barbun-cuda, Accessed on: Jan 2022.

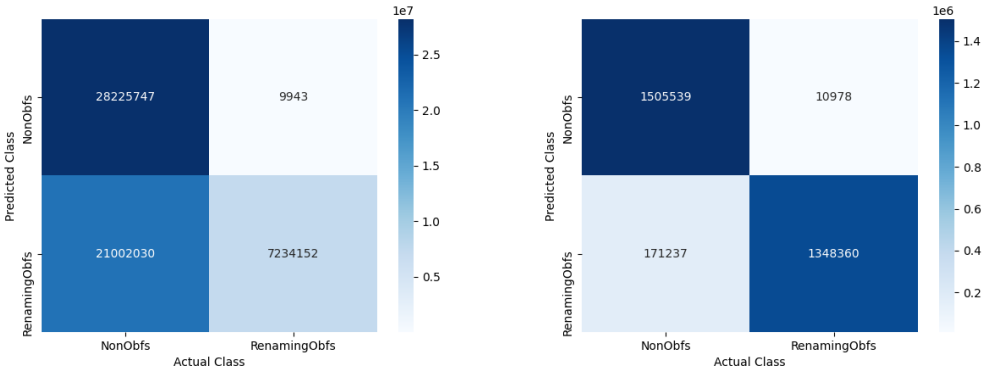
[119] M. Şırlancı, "Malicious code detection: run trace analysis by LSTM," Master's thesis, Middle East Technical University, 2021.

[120] [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/text/Tokenizer, Accessed on: Jan 2022.

APPENDIX A

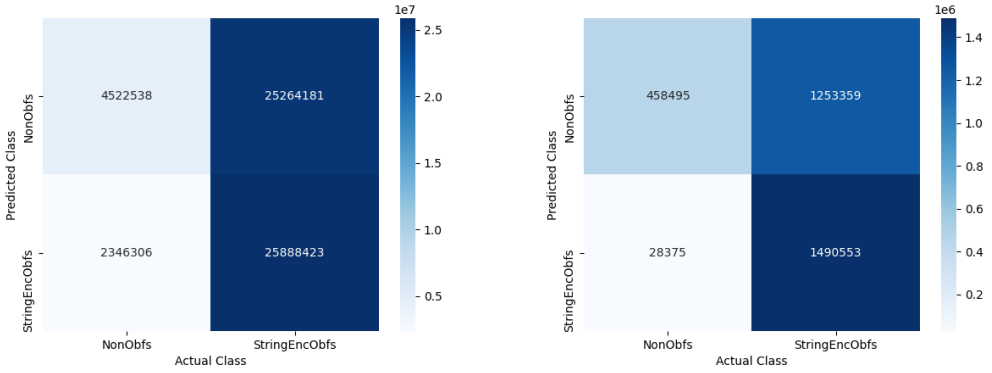
CONFUSION MATRICES

Confusion matrices of each detection unit for both the ISD and the MSD are given below:



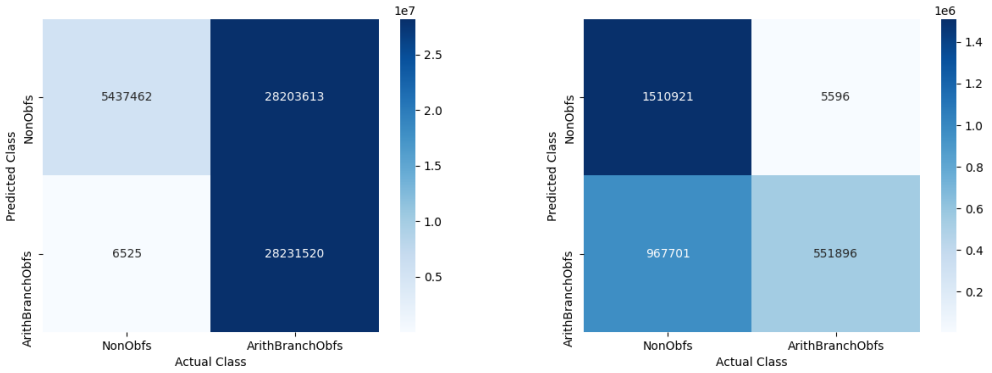
(a) The ISD (b) The MSD

Figure A.1: Confusion Matrices for Identifier Renaming



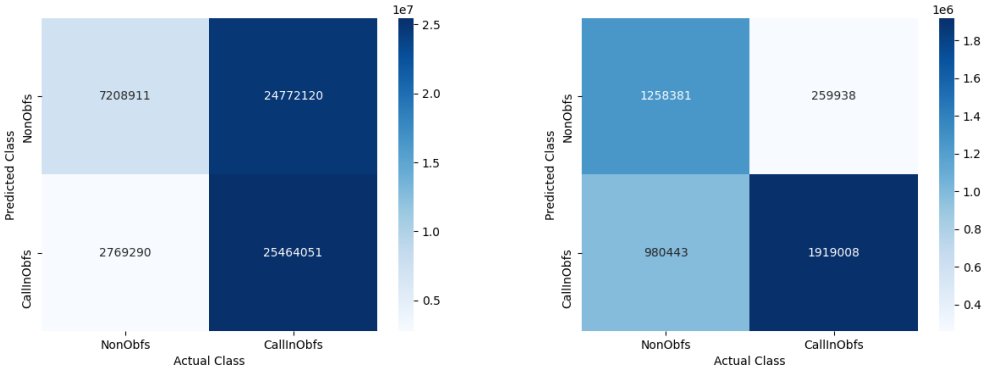
(a) The ISD (b) The MSD

Figure A.2: Confusion Matrices for Constant String Encryption



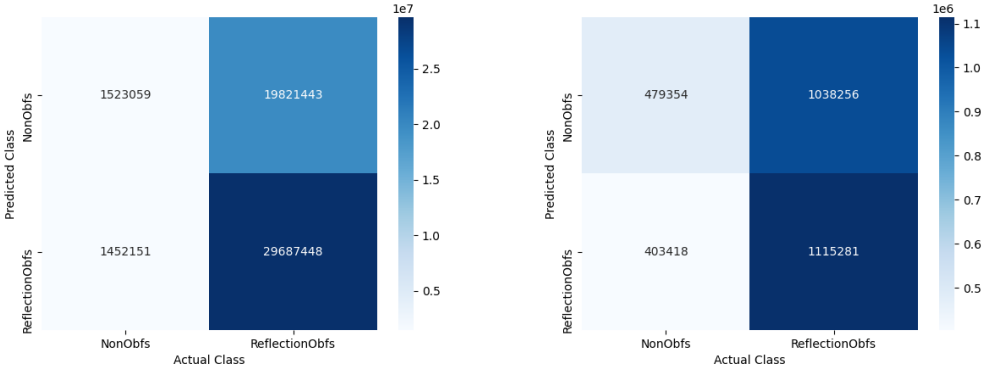
(a) The ISD (b) The MSD

Figure A.3: Confusion Matrices for Arithmetic Branch Insertion



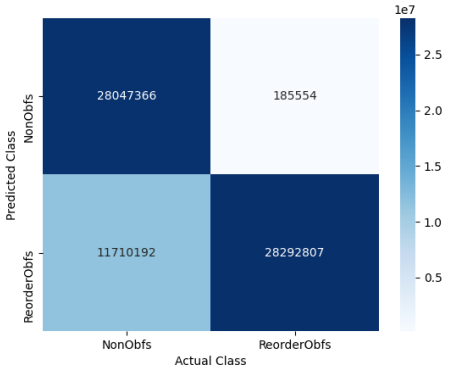
(a) The ISD (b) The MSD

Figure A.4: Confusion Matrices for Call Indirection

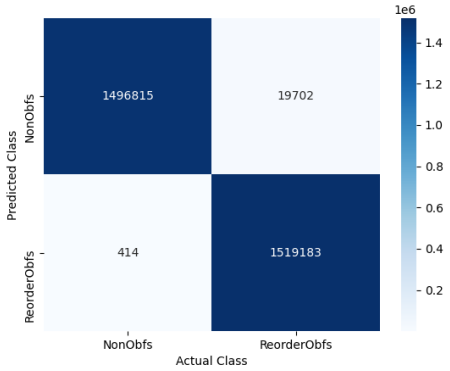


(a) The ISD (b) The MSD

Figure A.5: Confusion Matrices for Reflection

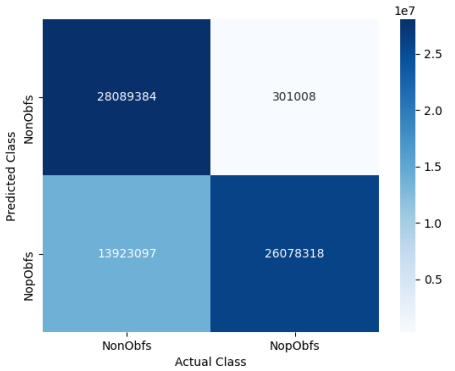


(a) The ISD

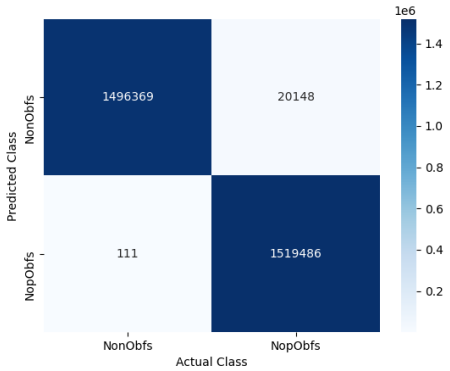


(b) The MSD

Figure A.6: Confusion Matrices for Reorder

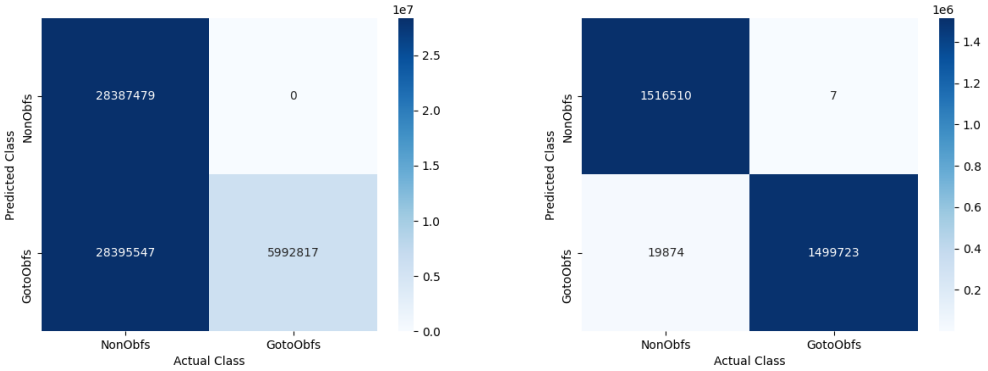


(a) The ISD



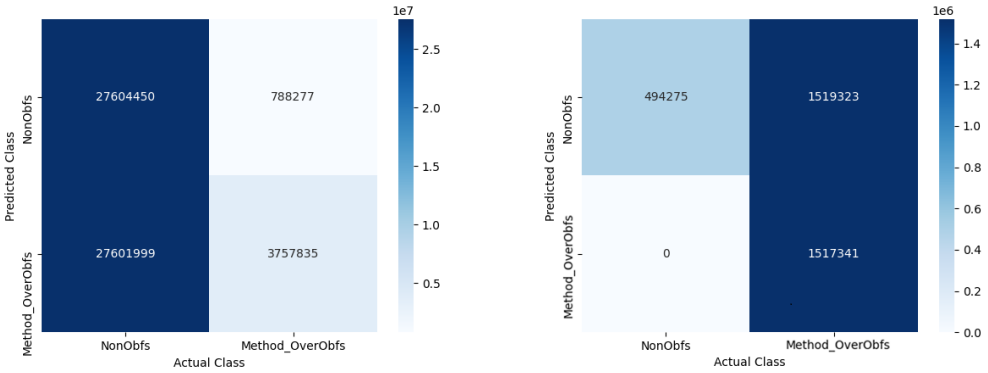
(b) The MSD

Figure A.7: Confusion Matrices for Nop Insertion



(a) The ISD (b) The MSD

Figure A.8: Confusion Matrices for Goto Insertion



(a) The ISD (b) The MSD

Figure A.9: Confusion Matrices for Method Overload