TRAPDROID: BARE-METAL ANDROID MALWARE BEHAVIOR ANALYSIS FRAMEWORK

by
HALİT ALPTEKİN

Submitted to the Institute of Engineering and Natural Sciences
in partial fulfilment of
the requirements for the degree of Master of Science

Sabancı University
July 2021

TRAPDROID: BARE-METAL ANDROID MALWARE BEHAVIOR ANALYSIS FRAMEWORK

Approved by:

Date of Approval: July 14, 2021

ABSTRACT

TRAPDROID: BARE-METAL ANDROID MALWARE BEHAVIOR ANALYSIS FRAMEWORK

HALİT ALPTEKİN

COMPUTER SCIENCE AND ENGINEERING MSc THESIS, JULY 2021

Thesis Advisor: Prof. Albert Levi

Thesis Co-Advisor: Prof. Erkay Savaş

Keywords: cyber security, dynamic analysis, mobile malware, android

In the realm of mobile devices, malicious applications pose considerable threats to individuals, companies, and governments. Cyber security researchers are in a constant race against malware developers and analyze their new methods to exploit them for better detection. In this thesis, we present TRAPDROID, a dynamic malware analysis framework mostly focused on capturing unified behavior profiles of applications by analyzing them on physical devices in real-time. Our framework processes events which are collected from system calls, binder communications, process stats, and hardware performance counters. Afterwards, it combines them into a simple, yet meaningful behavior format named UBF (Unified Behavior Format) using UPL (UBF Processing Language) scripts. We evaluated our framework's accuracy and performance on the up-to-date malware dataset, which contains widely-known variants, custom crafted malicious applications, 0-day, and 1-day samples. The framework is easy to use, extensible, fast, and providing high accuracy in malware detection with relatively low overhead.

ÖZET

TRAPDROID: ZARARLI ANDROİD UYGULAMALARININ GERÇEK CİHAZLAR ÜZERİNDE DAVRANIŞSAL ANALİZİ

HALİT ALPTEKİN

BİLGİSAYAR BİLİMLERİ VE MÜHENDİSLİĞİ YÜKSEK LİSANS TEZİ, TEMMUZ 2021

Tez Danışmanı: Prof. Dr. Albert Levi

Tez Eş-Danışmanı: Prof. Dr. Erkay Savaş

Anahtar Kelimeler: siber güvenlik, dinamik analiz, mobil zararlı yazılım, android

Zararlı mobil uygulamalar; devletlere, şirketlere ve son kullanıcılara yönelik ciddi bir tehdit oluşturmaktadır. Siber güvenlik uzmanları da bu tarz zararlı uygulamaları sürekli olarak analiz edip daha iyi bir tespit sistemi ortaya çıkarmaya çalışmaktadır. TRAPDROID ismini verdiğimiz çalışma, zararlı yazılımları dinamik olarak davranışsal analizini gerçek cihazlar üzerinde gerçekleştirebilmektedir. Bu platform, sistem çağrıları ile birlikte uygulamaların süreçler arası iletişimini ve donanımsal performans metrikleri harici herhangi bir bağımlılığı olmadan toplayabilmektedir. Tüm bu toplanan veriler ve metrik değerler, özel olarak yazılmış betikler (UPL - UBF Processing Language) kullanılarak davranışsal bilgiler içeren özel bir formata (UPF - Unified Behavior Format) dönüştürülmektedir. Geliştirdiğimiz zararlı yazılım tespit sisteminin başarısını ölçebilmek için kullanılan tanınmış veri kümeleri, özel olarak geliştirilen zararlı yazılımlar ve henüz daha sınıflandırılması yapılmamış uygulamalar ile zenginleştirilmiştir. Projemiz geliştirilmeye açık olmasının yanı sıra, hızlı ve yüksek bir başarı oranı ile mobil cihazlardaki zararlı yazılımları gerçek zamanlı olarak tespit edebilmektedir.

ACKNOWLEDGEMENTS

First of all, I would like to express my gratitude to Prof. Albert Levi, my advisor and one of the significant contributors to this thesis, with my co-advisor, Prof. Erkay Savaş. His supervision has motivated me immensely. The feedback he has given me for the past couple of months has been priceless, and I am incredibly grateful for his contributions to this thesis. I would also like to thank my co-advisor, Professor Erkay Savaş, for his timely and actionable feedback, and I am happy to have had the chance to work alongside him. I also prolong my gratitude to the jury members, Prof. Bülent Yener, Prof. Yücel Saygın, Assoc. Prof. Cemal Yılmaz, for participating in my jury, reviewing my thesis, and providing feedback. Finally, I am grateful for the endless support and guidance of Can Yıldızlı.

to my mother, wife, family, and mr. kernel

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATONS

IPC Inter-process Communication 2, 5, 16, 50

JNI Java Native Interface 7

JSON JavaScript Object Notation 25

LKM Loadable Kernel Module 1

ML Machine Learning 32

MWR Malware Ratio 2, 35, 39, 42, 43, 45, 48, 59

OEM Original Equipment Manufacturer 3, 21

PCAP Packet Capture 25

PID Process Identification Number 28

PMU Performance Management Unit 1, 2, 17, 50

ReLu Rectification Function 45

RF Random Forest 39

SMS Short message service 6, 9, 25, 29, 30, 34, 56

SVM Support Vector Machines 32, 39, 43

TBOX TRAPDROID Box 18, 19

TF Term Frequency 36

TF-IDF Term Frequency–Inverse Document Frequency 14, 36, 52

UBF Unified Behavior Format 2, 27, 28, 29, 32, 36, 37, 41, 43, 47, 52, 59

UDP User Datagram Protocol 1

UI User Interface 52

UID User ID 4, 17

UPL UBF Processing Language 2, 25, 26, 27, 28, 30, 31

URL Uniform Resource Locator 9

VPN Virtual Private Network 58

XMR Monero 10

1. INTRODUCTION

As Android[1] malware analysis and detection systems evolve, attackers also adapt their methodologies by leveraging detection and analysis techniques. Modern malware samples can detect the presence of a virtualization environment or instrumented functions, which allow them to evade detection (Jing et al., 2014). Besides, they use techniques like obfuscation, encryption, or dynamic loading to evade static analysis (Aslan & Samet, 2020). They also use anti-emulation and anti-debugging techniques to evade dynamic analysis (Park et al., 2020). There is a growing body of literature that recognizes the importance of using bare-metal environments for the detection of the evasive malware (Bulazel & Yener, 2017).

In this work, we present TRAPDROID[2], a scalable, dynamic malware analysis framework that is able to analyze a given application in real-time on physical devices. TRAPDROID is divided into two components: *driver* and *server*. The *driver* is a LKM (Loadable Kernel Module) that is responsible for interception of system calls, binder communications, collection of PMU events, and statistical information of processes collected from kernel structures like `task_struct`.[3] The raw events collected by the *driver* are securely transmitted to a server using custom UDP[4] protocol. All of the pre-processed events are populated into an advanced analysis engine using state-of-the-art machine learning techniques to differentiate whether a given application performed malicious activity or not. Our platform shows the results through a dashboard and allows users to involve in the analysis phase by triggering (stimulating) malicious activities of the target application. In order to reveal more hidden behavior, we employ the UI-coverage based approach with a depth-first search on an activity graph.

---

[1] https://www.android.com/what-is-android/

[2] The preliminary version of TRAPDROID has been published as a conference paper (Alptekin et al., 2019).

[3] https://tldp.org/LDP/lki/lki-2.html

[4] https://datatracker.ietf.org/doc/html/rfc768

TRAPDROID converts all pre-processed logs into UBF (Unified Behavior Format) by executing UPL (UBF Processing Language) scripts. This behavior profile format is simple yet adequate to represent the dynamic behavior of an application. The pre-processed logs contain the following information:

- a selected subset of system calls such as `open`, `close`, `socket`, `connect` with returned value of Linux kernel;
- broadcast events such as `SMS_RECEIVED`, `SCREEN_ON`, `SCREEN_OFF` emitted by system process through `binder`;
- IPC communication between target application and `binder`;
- PMU events of target process such as L1D-access, L1I-miss, instruction, branch-prediction count, etc;
- accounting information of target process such as `maj_flt`, `min_flt`, `stime`, `utime`, `read_char`, `write_bytes`, etc.

Our experiments achieved 98.41% accuracy using our CNN model with multiple embedding layers on a balanced (MWR=0.5) dataset containing 6000 applications collected from different research datasets and private resources. Furthermore, widely-accepted machine learning algorithms and deep-learning models also provide higher than 97% accuracy and F-1 score, which is enough to build a successful detection system.

The main contributions of our work are:

- A publicly accessible, open-source Android malware analysis platform and up-to-date dataset;
- Proposing a novel unified behavior format to reconstruct the activities of the target application;
- Testing the performance of the widely accepted text-classification approaches and proposing a novel CNN network;
- Analyzing the MWR and behavior coverage (λ) metrics over detection performance;
- Demonstrating that low-level features can be effectively used in Android malware detection, bridging the semantic gap between low-level features and malicious behavior;
- Comparing our findings with the existing researches.

2. BACKGROUND

This chapter provides a general overview of the Android applications, operating system details, and essentials of the malware detection methodologies.

2.1 Android Operating System

Android is a mobile operating system specifically optimized for resource-constrained devices and developed by Open Handset Alliance[1] with the commercial support of Google[2] since 2008. It has been the top-selling mobile operating system on smartphones since 2011. OEMs (Original Equipment Manufacturers), chip-makers, carriers, and application developers play a vital role in developing the operating system and core frameworks. The overview of the Android architecture is shown in Figure 2.1.

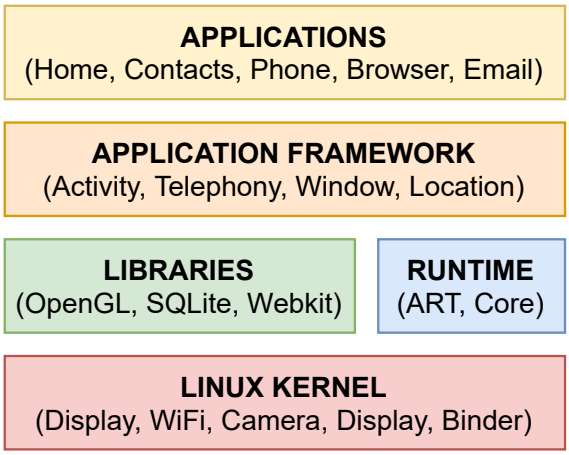

Figure 2.1 Overview of the Android Architecture.

[1] https://www.openhandsetalliance.com/

[2] https://developers.google.com/android

The Android architecture consists of four layers. The first layer at the bottom is the Linux Kernel, responsible for the file system, task scheduling, memory management, socket handling, and other critical tasks. The upper level employs system-wide libraries, Android Runtime (ART)[3] and its predecessor, Dalvik virtual machine (DVM), which provide core functionalities and a runtime environment for Android applications. Most of the system services in the application framework layer are written in the Java programming language. The Android applications are running in the last level known as the application layer.

2.1.1 Linux Kernel

Linux kernel is a monolithic, UNIX-like, free, and open-source operating system kernel. Android patched the mainline Linux kernel to add new components[4] such Binder and AShmem. Each Android application has a unique UID value assigned by Linux to prevent applications from interacting directly between each other's data and processes. However, they invoke the Binder driver in case of inter-process communication. Figure 2.2 depicts the call graph of a sample binder transaction.

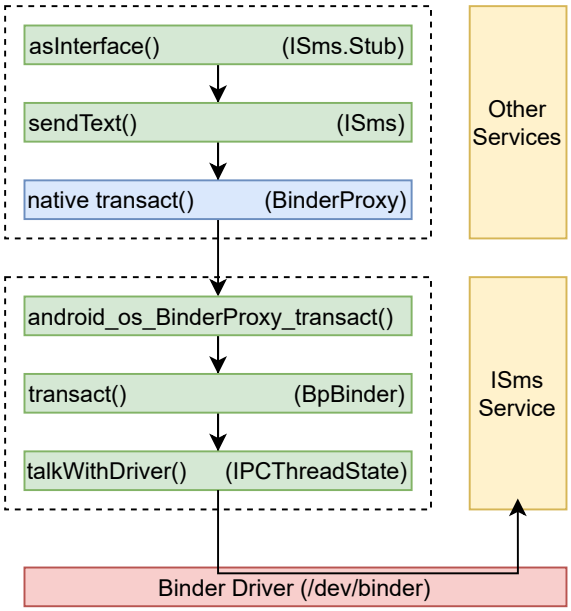

Figure 2.2 Call graph of sample binder transaction.

[3]https://source.android.com/devices/tech/dalvik

[4]https://elinux.org/Android_Kernel_Features

2.1.2 Binder

Binder is a kernel driver responsible for handling IPC (Inter-process communication) calls. These calls handle data transfer between applications through a protected environment. Applications make use of ioctl system call in order to execute IPC calls of the Android operating system. In order to receive requests and send responses, objects define interfaces using Android Interface Definition Language (AIDL).[5]

2.2 APK File Structure

The Android applications use APK (Android Package Kit) as their main application format. This format contains required files such as AndroidManifest.xml, icons, signatures, assets, native libraries, resources, and the executable classess.dex file, which is compiled for DVM (Dalvik Virtual Machine). AndroidManifest.xml is the most important file among them since it contains many definitions such as package name, permissions, services, receivers, and providers. Figure 2.3 represents the inner structure of the APK.

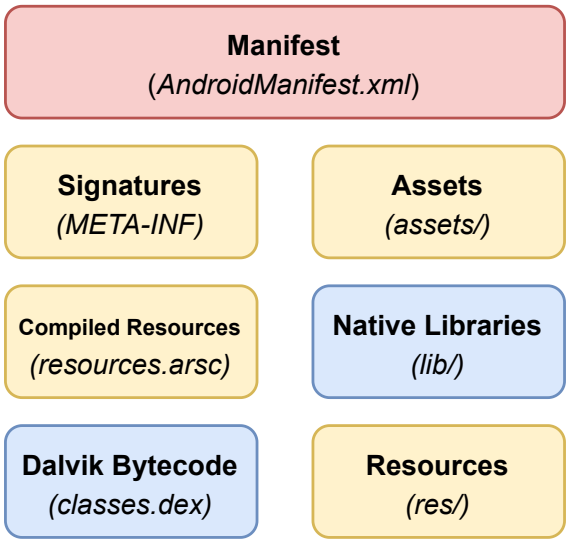

Figure 2.3 APK file structure.

[5]https://developer.android.com/guide/components/aidl

2.2.1 Permissions

Mobile application developers set required permissions in the manifest file to allow their software to perform privileged operations such as sending or receiving SMS, accessing the internet, dialing, or setting wallpapers. The Android operating system provides permissions with protection levels categorized as signature, system or signature, normal and dangerous. Regular permissions (normal protection level) do not pose any risk to the user's privacy. Therefore, the operating system grants access automatically if those sorts of permissions are listed in the manifest file. On the contrary, dangerous permissions enable applications to access confidential information stored on the device. However, users can grant or revoke permissions upon the installation. Android 6.0 or newer versions have this facility of granting or revoking permission.

Table 2.1 Significant permissions for malware detection.

Permission Name	**Description**
READ_CONTACTS	Read-access to contacts
MANAGE_ACCOUNTS	Manage the account list
ACCESS_COARSE_LOCATION	Obtain the current device location via GPS
INTERNET	Full internet access
WRITE_SETTINGS	Modify global device settings
INSTALL_PACKAGES	Install new applications
ACCESS_NETWORK_STATE	View network/connectivity status
READ_EXTERNAL_STORAGE	Access to external storage (SDcard)
SEND_SMS	Send a short message to destination
WRITE_APN_SETTINGS	Update APN settings
GET_ACCOUNTS	Discover known accounts

Threat actors demand developer-defined or self-defined permissions to gain access to the victims' private information. It is expected for the application to behave in a way, which requires these set of permissions during the execution. For instance, if any application defines `READ_CONTACTS` permission, we expect that application invokes to regarding API call to obtain contact list. These expectations enable us to develop behavior coverage metric to assess the performance of the stimulation engine. Table 2.1 represents several significant permissions (Li et al., 2018) requested by malware samples.

2.2.2 Intents

An Intent is an unique messaging object which enables applications to request an action from another application. The requested action includes starting an activity or service and delivering a broadcast. There are two types of intents: implicit and explicit. While implicit intent can interact with other applications within the operating system, explicit intent is developed for message passing between the same application components.

2.2.3 Native Libraries

Native libraries are widely used in Android applications as they allow developers implementing functionality with C/C++ or even with assembly language. In some use cases, it is advantageous to use these lower-level programming languages, not only due to better performance but also because it is easy to re-use the code that had been developed for other platforms.

Native libraries are compiled into .so (shared object) files for each targeted platform architecture (x86, x86-64, armeabi etc.) and they are dynamically loaded into memory when the application is launched. There is a communication interface called Java Native Interface (JNI) to allow calling functions in native libraries from Java layer or vice versa.

It is known that Android malware can utilize native libraries in order to hide the functionality running in the background by means of obfuscation which is usually harder to reverse engineer when compared to Dalvik Bytecode.

2.2.4 Dalvik Bytecode

Android apps are written in Java language and run in Android Runtime (ART) environment. In this environment, unlike the discontinued project Dalvik, ART converts the application's bytecode into native instructions.

2.3 Android Malware Ecosystem

There are different types of malware threats affecting the Android operating system. In this section, we briefly present the most common types of these malicious applications. Table 2.2 shows some real-world examples of different malware types.

Table 2.2 Real-world examples for each malware category.

Category	**Variants**
Adware	Batmobi, Ewind, Hummingbird, Mobisec, Loki
APT	Confucius, Pegasus, FinSpy, Monokle, Tangelo
Banking Malware	Alien, Cerberus, Hydra
Cryptocurrency Miner	HiddenMiner, ADB Miner
Ransomware	Congur, Masnu, Fusob, Jisut, Koler, Lockscreen
Scareware	Avpass, Mobwin, Fakeapp
SMS Worm	Selfmite, Flubot, Goodnews, Samsapo, Filecoder
Spyware	Spynote, Stalkerware, Spydealer, Smsthief, Spyagent

2.3.1 APT

An advanced persistent threat (APT) is a highly-skilled threat actor who mainly develops malicious applications to increase their privileges on the operating system and hide their activities to maximize persistence. As a result, they may stay hidden for months without the user noticing any suspicious activity. Nation-state threat actors regularly develop mobile APT malware usually backed up with 0-day exploits. This type of mobile malware is generally aimed at high-profile targets and is not easily detected by modern AV (Antivirus) software. For instance, Pegasus[6] is a notorious APT sample developed by NSO Group[7] to allegedly tackle crime and terrorism.

[6] https://info.lookout.com/rs/051-ESQ-475/images/lookout-pegasus-android-technical-analysis.pdf

[7] https://www.nsogroup.com/about-us/

2.3.2 Adware

Adware is another type of mobile malware that generally does not show any malicious behavior but is designed to pop up ads and trick users into profiting by selling products or increasing website traffic. These applications mainly slow down the operating system and often try to direct users to install other variants to maximize the gain of the threat actors.

2.3.3 SMS Worm

One of the most significant examples of Android malware is the SMS Worm. These malicious applications are a popular type of mobile malware that tricks users by spreading themselves via SMS. Since many users are well adapted to using SMS technology, the spreading speed of malware is almost unbeatable. Furthermore, this type of malware also utilizes the contact list of the victim's device to trick the victim's close circle, thus making the spreading much faster. Figure 2.4 shows the sample SMS contains the malicious file URL sent by malware during the COVID-19 pandemic.

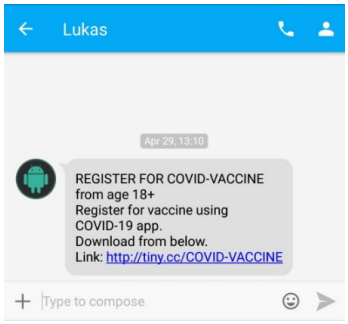

Figure 2.4 Sample text message sent by SMS Worm.

2.3.4 Banking Malware

One of the most common types of threats affecting the finance sector is the use of banking malware. The banking malware is a specially crafted malicious software to steal the user's credentials and other confidential information. These applications generally spread through large phishing attacks, smishing, or social engineering.

2.3.5 Scareware

The purpose of scareware is to trick users into downloading and paying for unnecessary software. The application itself does not necessarily contain any malicious code. Nevertheless, they are designed to mislead users to feel obligated to pay for specific software to maintain their devices. As these applications do not have any malicious code in themselves, they can be downloaded from public markets.

2.3.6 Spyware

Spyware is a generic term to represent any application that is capable of exfiltrating certain private information such as the browsing history, messages, contacts, or other confidential information stored on the device without the user's consent. Spyware can be installed as a standalone application or it may well be included as a module of another benign application.

2.3.7 Ransomware

Ransomware is a specially developed application that prevents users from accessing services and data on their devices by encrypting or disrupting them until a considerable ransom amount is paid. Ransomware has gained popularity in recent years due to the rise of crypto coins and anonymous payments. These types of attacks mostly have catastrophic consequences and their damage is almost irreversible.

2.3.8 Cryptocurrency Miner

While not as popular as their desktop counterparts, mobile malware samples mining cryptocurrencies like BTC, ETH, or XMR in the device are constantly increasing. This type of malware is generally bundled with benign applications as users need to keep using the application during the mining phase. However, mining on phones is not as profitable as PCs and needs more technological development.

2.4 Malware Detection

Malware detection can be separated into two main categories: *static* and *dynamic* analysis. During recent years, malware developers have found multiple methods (Bello & Pistoia, 2018) to evade static analysis, which shaped the working dynamics of modern anti-virus products. As malware gained more capability in hiding its tracks and evading detection methods, dynamic analysis (Yan & Yin, 2012) gained immense importance to cope with the growing threat. However, recent solutions also became ineffective to detect sophisticated malware threats, which opens up using hybrid analysis (Kabakus & Dogru, 2018) and bare-metal frameworks (Mutti et al., 2015) in detection software.

2.4.1 Static Analysis

There is a considerable amount of research on static malware analysis that utilizes the static properties (Feng et al., 2014) of the executable and provides high scalability with relatively low computation resources without executing them. The several properties of the APK file, intents, permissions, API calls, information flow, and opcodes are essential for the static analysis.

The permission set extracted from the manifest file is a primary feature resource of the researchers. Static analysis approaches heavily dependent on the defined permissions. Nevertheless, application developers tend to extravagantly define permissions in a manifest file even the application does not need all of the permissions. Furthermore, APT or advanced malicious applications might not declare any permission to carry out their actions. They can use the privilege escalation exploits to bypass the security enforcements of the defined permissions.

One of the most popular static features is the source code of the program, if available. Otherwise, the binary file needs to be disassembled or decompiled. Researchers utilize the semantic representation of the source code by building control-flow or data-flow graphs and sequences of API calls. However, the actual malicious code can be downloaded from external resources or generated during the application runtime. Therefore, the detection system built on top of source codes can be easily evaded by dynamic loading (reflection) techniques.

2.4.2 Dynamic Analysis

There are several approaches proposed to overcome the limitations of the static analysis. Unlike static analysis, dynamic analysis is carried out in controlled (emulator) or bare-metal (real device) environments. Most of the techniques utilize the system calls, binder transactions, resource consumption, and network traffic. Since the dynamic analysis relies on runtime behavior, not just the source code or static properties, it is less prone to dynamically loaded malware.

Upon executing applications, some of the variants can evade the analysis environment using several bypass techniques. In order to observe the activities of the malicious applications firmly and block the evasive attacks, execution environments need to be appropriately configured. The bare-metal platforms provide a more suitable environment instead of emulators.

Analyzing the system call sequences of the application under inspection is one of the most popular dynamic analysis approaches. In this approach, detection systems can be configured for intercepting the system calls between application and kernel. Since the kernel modifications are complicated and challenging to apply different versions due to compatibility issues, most researchers use the `strace`[8] tool to capture the system calls. The `strace` tool utilizing `ptrace`[9] system call and can be easily detected by evasive malware.

Alongside the system calls, API calls invoked by applications are also helpful for malware detection. Since the Android operating system cannot allow direct access to resources such as contacts, messages, or other private information, applications need to call related API calls. For instance, adding a new device admin, obtaining the whole message database, reading notifications, or dialing numbers can be considered suspicious behavior.

The network traffic of the Android application can be used to identify malware. The entropy of the domain name, packet counts, protocols, destination ports, or reputation of the IP addresses are example features extracted from network activity. However, there is a widely accepted phenomenon called the ephemerality of malware. Accordingly, the command and control servers of the malware shut down in a short period, and it needs to be instantly analyzed upon availability.

---

[8] https://man7.org/linux/man-pages/man1/strace.1.html

[9] https://man7.org/linux/man-pages/man2/ptrace.2.html

2.5 Related Works

There has been significant research on using the bare-metal framework for detecting Android malware in recent years. This section present related work and discuss detection methodologies of recent researches.

Crowdroid (Burguera et al., 2011) is an Android malware analysis system that uses ptrace to log and analyze an application's system calls. It can collect the system-wide device information, installed application list, and system call sequences of the application under the analysis. Upon collecting the required information, the remote server applies clustering algorithms to identify malicious applications.

Similarly, DroidTrace (Zheng et al., 2014) is a standalone classification system that utilizes ptrace system call within the device itself to identify malware. It uses a dynamic library loading technique to monitor selected system calls. Moreover, the authors developed a forward-execution mechanism that can disassemble and repack the samples to provide high behavior coverage. However, both of these tools are highly dependent on the usage ptrace system call, and evasive malware samples can identify whether their active processes are under analysis or not.

On the other hand, DroidScope (Yan & Yin, 2012) is a malware detection project that monitors the samples in the emulated environment without utilizing the ptrace system call. It implements taint-tracking and enables introspection at different layers of the platform. However, the platform is based on QEMU[10] and cannot mimic the properties of physical devices.

BareDroid (Mutti et al., 2015) uses the power of a bare-metal platform. It enables to use of real devices for Android malware analysis. Nevertheless, instead of using their platform for practical malware analysis, the authors explained the feasibility and procedures of running a bare-metal system for detection. We opted to use BareDroid's threat model in our project.

Some researchers (Canfora et al., 2015) also apply machine learning algorithms to detect malicious applications in real devices. Instead of the classical approach, they consider the limited number of system calls to bridge the semantic gap between behavior and system calls. However, the Android operating system constantly evolves and adding new functionalities to its platform. Therefore, the limited number of system calls is inadequate to represent some of the malicious behaviors.

[10] https://www.qemu.org/

In addition to traditional machine learning approaches, several deep-learning-based detection systems also proposed dynamic analysis methods. Deep4MalDroid (Hou et al., 2016) is a fully automatic malware detection system based on an emulator. It builds the weighted graph of system calls invoked by the sample application and populates the generated graph into a deep learning model containing stacked auto-encoders.

In order to encode the behavior sequence, researchers (Xiao et al., 2019) applied Long Short-Term Memory (LSTM) language model to system calls. They developed two models for malware and benign families to calculate the similarities between them. Although their promising results, their detection system is built on top of ptrace system call and prone to evasive attacks.

The Convolutional Neural Network (CNN) is another deep learning model specially developed for image processing problems. Researchers (Abderrahmane et al., 2019) developed a custom CNN architecture to detect Android malware samples using system calls invoked by samples under the analysis. The CNN network expects the dependency matrix to determine whether the sample is malware or benign. In our research, we treat system call sequences as a text and employ TF-IDF vectorization before populating the input matrix into the CNN network.

Most of the previous work has prevalent limitations such as the usage of outdated Android versions (Kouliaridis et al., 2020), lack of dataset information and publicity (Qiu et al., 2020), susceptibility to evasion (Jing et al., 2014), imbalanced variant distribution (Wang et al., 2020), or producing low behavior coverage due to the ephemerality of the malware. In order to overcome these limitations, we combine widely-known datasets (Arp et al., 2014; Mahdavifar et al., 2020) with 0-day and 1-day samples which emerged during our research. Upon analyzing the behavior of these samples using the bare-metal platform called TRAPDROID, we tested certain classification algorithm's performance using TF-IDF with n-grams and novel deep-learning architectures.

3. TRAPDROID FRAMEWORK

Our study required a fully implemented bare-metal framework to test our methodology and validate our results. This section presents our framework in detail by covering all distinct stages, including data collection, stimulation, and automatic state restoration.

3.1 Data Collection

The TRAPDROID framework consists of two main components: *driver* and *server*. The *driver* gathers the raw activity log from the kernel level and sends them to the remote server using the `netpoll`[1] mechanism. As a next step, API converts logs into a meaningful yet straightforward event stream. The human analyst can track the actual progress of analyzing the suspected application from a user-friendly dashboard. TRAPDROID is able to provide all activities in real-time using WebSocket technology and REST API to external systems for integration. Figure 3.1 depicts the overview of the framework.

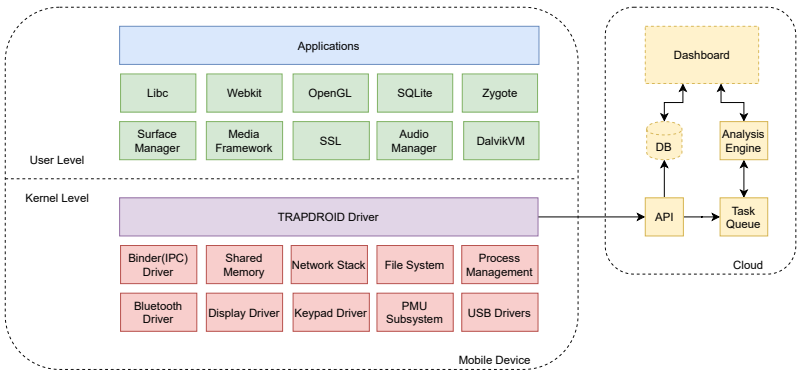

Figure 3.1 Overview of the TRAPDROID framework.

[1] https://wiki.ubuntu.com/Kernel/Netconsole

The *driver* module intercepts some selected system calls via `kretprobes` without altering the system call table when mounted to the system. We have selected 15 of the most crucial system calls to reconstruct the behavior of the analyzed application. In addition, the system filters `ioctl` system calls on the binder driver to intercept all IPC communications. TRAPDROID is implemented in a modular way to be extended easily to capture new system calls or other sorts of data such as memory access patterns, scheduler stats, etc. We evaluated most of the system calls and kernel functions to boost the detection system's performance, and we decided to use a compact subset of system calls, as shown in Table 3.1. As an alternative to our approach for intercepting system calls, this process is traditionally carried out by strace tools.

Table 3.1 Intercepted system calls.

File System	open, close, rename, unlink, mkdir, access, getdents
Process	fork, execve, clone, ptrace
Network	socket, connect
Privilege	chown, chmod

`strace` is a debugging tool capable of intercepting all system calls invoked by the target process, taking advantage of the `ptrace` system call. However, evasive malware samples can identify whether their active processes are under analysis or not. Therefore, they can bypass the dynamic analysis systems based on `strace`. For instance, malware developers may craft malware equipped with the native library, which can bypass strace-based analysis for the Android operating system. The below code is a motivational example of the traditional anti-debugging technique which thwarts ptrace-based tools.

```
void anti_debug() {
    int pid = fork();

    if (pid == 0){
        int ppid = getppid();
        int status;

        if (ptrace(PTRACE_ATTACH, ppid, NULL, NULL) == 0){
            waitpid(ppid, &status, 0);
            ptrace(PTRACE_CONT, ppid, NULL, NULL);
        }
    }
}
```

Listing 3.1 Motivational example of the anti-debugging.

In our framework, the driver identifies whether the current process is our target or not by looking at its UID value. This step is trivial as the Android operating system assigns a unique UID value to each installed application. When a system call occurs, the driver checks whether the current process's UID value is equal to our target UID. Then, the driver pushes a new event containing the information collected from `task_struct` and PMU into a circular queue. Another kernel worker processes items in the queue and delivers them to the remote server using `netpoll`.

The starting of Android applications can be captured by hijacking fork system calls of the `zygote` process. If the driver detects a new application executed in the operating system, it clears and enables the hardware performance counter of the target process for defined PMU events such as data cache access, branch-misprediction, instruction, etc. When the target application invokes the hooked system calls, the driver stores the exact values of hardware performance counters. Targeting PMU events can be configured or extended easily as represented in below code blocks using perf-subsystem of the Linux kernel.

```
static struct perf_event_attr pattr_cpu_cycles = {
    .type = PERF_TYPE_HARDWARE,
    .size = sizeof(struct perf_event_attr),
    .config = PERF_COUNT_HW_CPU_CYCLES,
    .exclude_user = 0,
    .exclude_kernel = 1
};


static struct perf_event_attr pattr_branch_misses = {
    .type = PERF_TYPE_HARDWARE,
    .size = sizeof(struct perf_event_attr),
    .config = PERF_COUNT_HW_BRANCH_MISSES,
    .exclude_user = 0,
    .exclude_kernel = 1
};
```

Listing 3.2 Sample PMU configurations.

As an alternative approach, there is another way to gather hardware or software PMU events of the system with the help of the `perf`[2] tool. However, we preferred the unified behavior analysis approach rather than relying on different applications to gather different metrics. Moreover, to analyze a suspicious application, we need to have more privilege on the operating system than the tracked application. Otherwise, evasive malware could bypass our analysis framework.

[2]https://perf.wiki.kernel.org/index.php/Main_Page

3.2 Environment

Recently, there is an increasing interest in the research of automated dynamic malware analysis systems running on physical hardware. As the main advantage of our proposed work is using the capabilities of a bare-metal environment, we have deployed our framework to a physical mobile device and analyzed various malware samples. Table 3.2 shows our analysis environment.

Table 3.2 Analysis environment.

Model	Sony Xperia XZ 2
OS	Android 10
Chipset	Qualcomm SDM845
Memory	4 GB
Storage	50 GB
Kernel	4.14.226-msm
Internet	Connected, no-restrictions
Cellular	Connected, without internet
PMU	Active

Most of the frameworks developed by researchers use old Android versions. Older Android versions lack many features and high percentage of new malware samples only works on modern Android versions. As a result, we targeted Android 10 in order to test the recent features of the Android operating system and be able to include the novel malware samples in our work. This selection enabled us to develop a sound, dynamic analysis framework and produced more relevant results in real-world scenarios.

Our analyzing environment consisted of two main components: real-device and TBOX (TRAPDROID-Box). While the real device responsible for executing the malware samples, TBOX manages the device and stores collected information appropriately in the database. Although we deployed TBOX as a mini-computer, it can be used in the cloud environment after minor modifications. It is developed using modern programming languages (Python, Flask) and scalable products (MongoDB, Elasticsearch) for further improvement.

3.2.1 TBOX

TRAPDROID is capable of sniffing the network communications while analyzing the target application. In order to capture network packets and provide an internet connection to the mobile device via WiFi, we created a protected network environment using TBOX. TBOX is a mini-server running with Arch Linux ARM on Raspberry Pi. In order to provide continuous charging and directly communicating through ADB, we connected our analysis device to TBOX via USB 3.0 with an external adapter. TRAPDROID utilizes `tcpdump`[3] software to capture all network packets and parses them using the `scapy`[4] library.

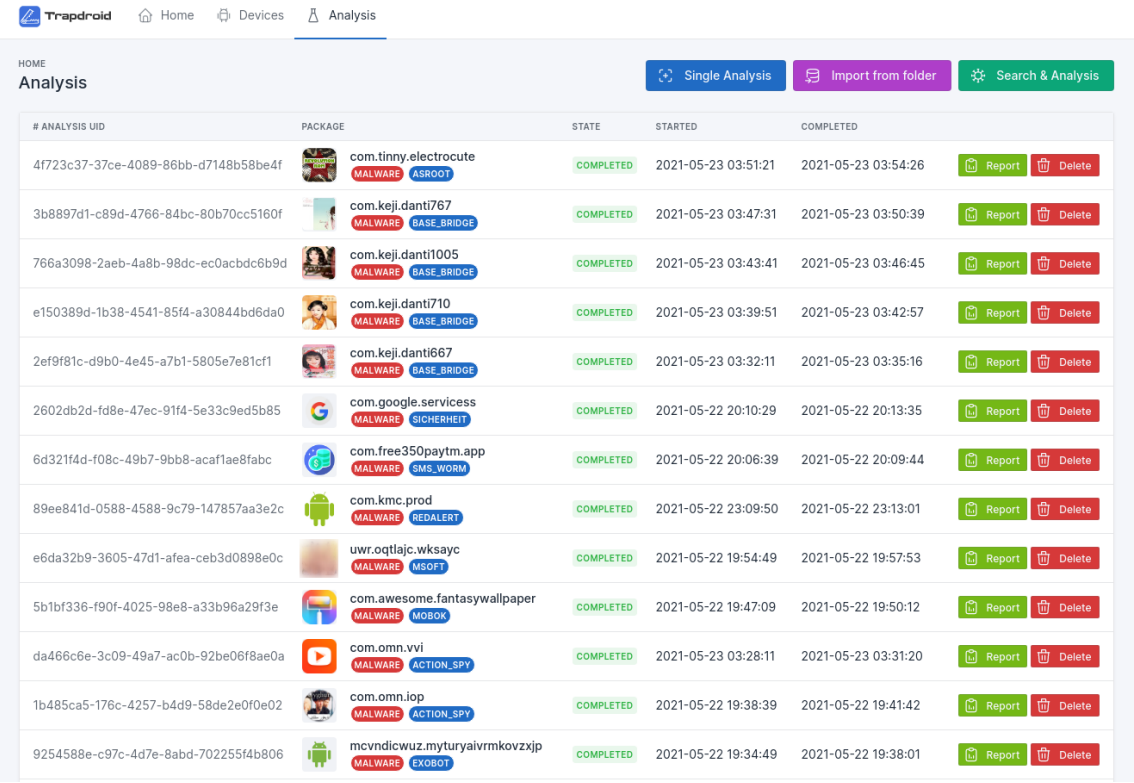

Figure 3.2 The dashboard of the TRAPDROID analysis engine.

Another functionality of the TBOX is to serve the Dashboard of the analysis engine. Figure 3.2 shows the analysis queue, which contains completed and active tasks. From this panel, analysts start a new analysis or obtain the generated report regarding the sample. Upon execution of the application, the analysis engine automatically determines the label using an active detection model.

[3]https://www.tcpdump.org/

[4]https://scapy.net/

3.2.2 Emulator vs Bare-metal

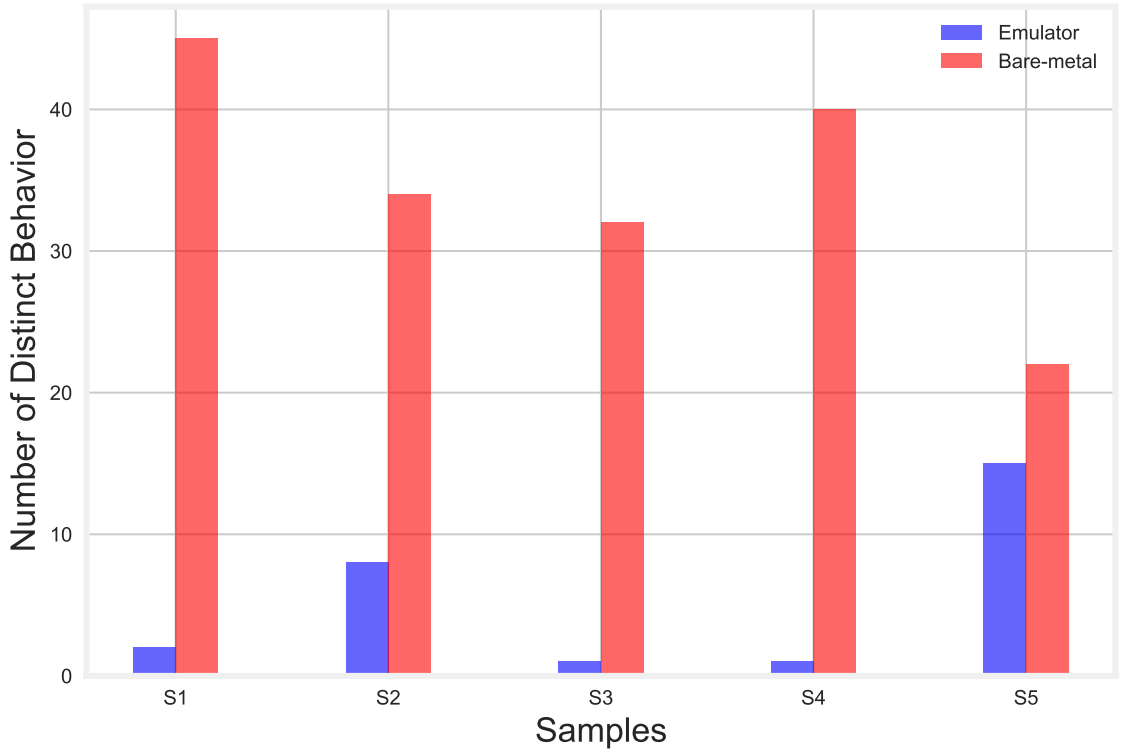

Figure 3.3 Behaviour comparison of the Emulator and Bare-metal environments.

To examine the effects of the bare-metal environment in terms of the number of revealed distinct behavior, we conducted an experiment with five different evasive malware samples, which belong to Ztorg, OBAD, and Hehe family. Our results can be seen in Figure 3.3. Four of the evasive malware terminated upon execution in the emulator either because they detected the emulation or crashed due to failed state, while the last sample (S5) failed at a later step. All executions were successful in the bare-metal framework, which allowed us to analyze the whole malicious activity of the samples. Although we tested our system with a relatively small dataset, numerous works provide similar outputs (Mutti et al., 2015). Therefore, bare-metal frameworks seem a more reasonable option for dynamic analysis rather than emulators.

3.3 Automatic State Restoration

In order to start analysis process using physical hardware, partitions of the mobile device should be properly organized. Our framework keeps a clean state of the `system` and the `userdata` partition in the Android sparse image format. These partitions always keep the pristine state of the device and cannot be altered during the execution in the analysis phase.

In case a modification is detected on the `system` partition, we use our previously saved state of the original `system` partition in our file system and flash it on the device accordingly. We opted to use the threat model, which is explained in more detail in BareDroid. The applications under the analysis can read/write to the `userdata` partition. Additionally, we rely on SELinux policies to keep our `system` partition safe from malicious applications. We have a validation mechanism to determine whether the `system` partition is altered or not. TRAPDROID analyzes the past activities of malware and decides to flash partition into the device via `fastboot`.

For the restoration, TRAPDROID boots the device into fastboot mode. The content of the `cache` partition is erased in the fastboot mode. Then, the original `boot` and `userdata` partition is flashed regardless of the malicious application's activity as in BAREDROID. `system`, `vbmeta` and OEM partitions are flashed as well if any threat is detected against other partitions' integrity.

Table 3.3 Procedure timings.

Step	**Time (seconds)**
Flash the `boot` partition	2.61
Flash the `vbmeta`, `dtbo` and other partitions	0.43
Restore `system` partition	39.09
Restore `userdata` partition	23.04
Boot the operating system	10.15
Execute the stimulation engine	180

TRAPDROID uploads the kernel module into the system and imports it after the ADB connection has been established successfully. It waits for the graphical user interface of the Android operating system to start the analysis process. After confirming every step during the preparation phase, we start our simulation engine.

The whole stimulation and state restoration process take roughly four minutes for a single application. Table 3.3 shows the required timings for each procedure.

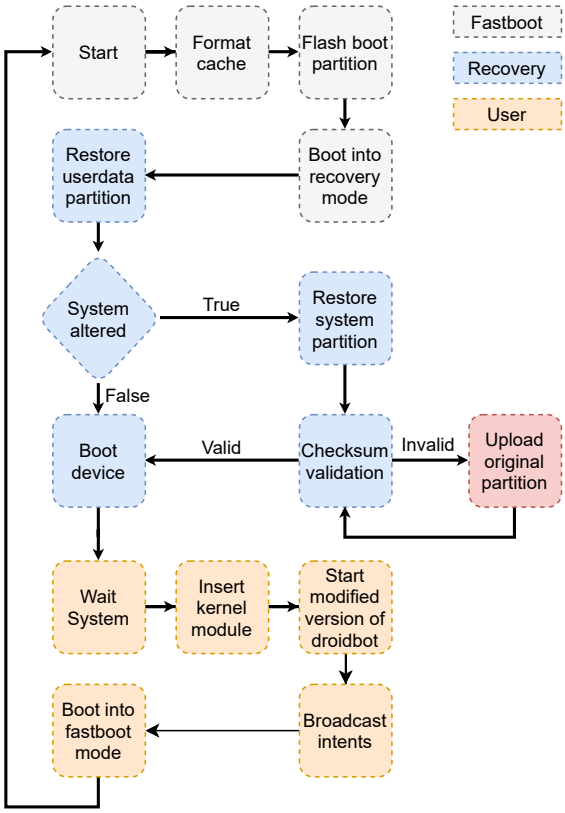

Figure 3.4 Automatic state restoration procedure.

We take advantage of the static analysis so as to uncover hidden behaviors of target application and reduce the analysis time. We have focused on neither how static analysis useful at malware detection nor to improve the static analysis techniques. We only leverage the static analysis report to improve the efficiency of the dynamic analysis. TRAPDROID extracts the statically defined broadcast receivers, which are expected by the target application from its manifest file. In addition to static receivers, our framework also finds the receivers which are registered dynamically. Our simulator engine uses this knowledge to broadcast more relevant intent actions in order to interact with the application. Figure 3.4 depicts the simplified representation of the automatic state restoration procedure.

3.4 Stimulation

Stimulation is the process of executing consecutive tasks for revealing more activities of the sample under analysis. While the triggered number of hidden activities increases the performance of the classification systems, it also reduces the probability of the overfitting problem.

In the research community, there are several stimulation engine options to trigger the hidden behaviors of the malware sample. These engines emulate the user interactions like clicking the button, scrolling or populate the test data to text fields to increase the activity coverage. In this work, we have used an improved version of the DroidBot (Li et al., 2017) as a primary stimulation engine. To increase the efficiency, we have altered the life-cycle of the DroidBot and added some new features like broadcasting several intents, auto-enabling accessibility, and notification services. Besides the automatic stimulation, we also employ manual stimulation when the engine cannot reveal activities sufficiently. However, barely a small portion of samples needed a manual inspection during our research. Figure 3.5 depicts the our stimulation acceptance methodology.

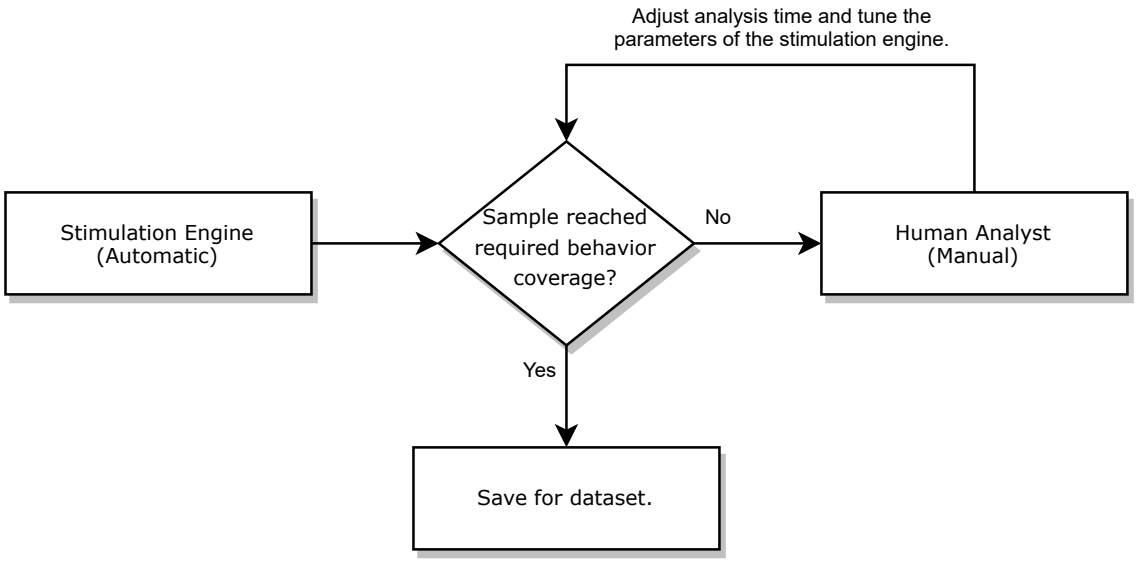

Figure 3.5 Stimulation acceptance methodology.

In our experience, the malicious behavior of the samples can principally be triggered by our stimulation engine. However, further investigation and experimentation of how to effectively trigger the malicious behavior of an application is another research area that needs to be addressed.

3.4.1 Human Analyst vs Monkey

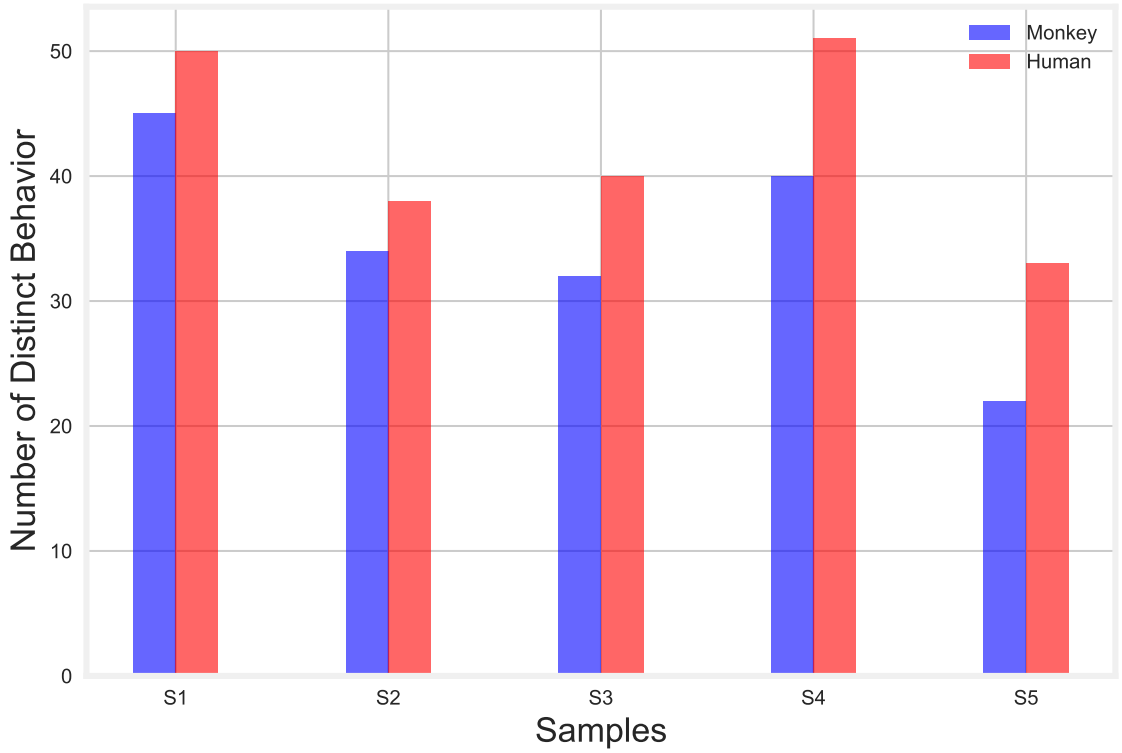

Figure 3.6 Stimulation comparison of the Monkey and Human Analyst.

Most of the malware need some form of interaction for stimulation, such as inputs from the user or the occurrence of certain events. In order to test the efficiency of the proposed system, we have measured the number of triggered functions using the `monkey`[5] tool and an expert analyst. In Figure 3.6, we show the number of distinct behavior actions of the malware triggered with the help of an expert analyst and the `monkey` tool. The results show that an expert analyst made it possible to observe 22.5% more actions on average than the automated counterpart. Triggering more functions by an expert analyst shows the advantages of using bare-metal environments for malware analysis as the number of observable functions differs significantly. In order to improve the efficiency of the `monkey`, we revised the DroidBot by adding valuable features like enabling accessibility or notification services.

[5]https://developer.android.com/studio/test/monkey

3.5 UBF Processing Language (UPL)

In order to process raw events, we developed a DSL (Domain-specific Language) named UPL (UBF Processing Language). Although it is not an actual Turing-complete programming language, even it has grammar, developers can include Python scripts that allow UPL to access third-party libraries like pattern processing, machine learning, or signal processing. This combination significantly reduces the time of need for the development of UPL scripts. Consequently, we can consider the UPL as a compound programming language with Python and JSON. UPL scripts can be extended by defining new blocks which have different intents as follow:

Behavior: In order to reconstruct the target applications' activities, we mainly declare new behavior blocks in the UPL scripts, which include system and API calls. We use a specific behavior block to accept all API calls with their respective API name and target code located in the binder transaction.

Annotation: We delicately declared annotation blocks to enhance the output of the behavior blocks. The extension of the file tried to open, remote port of the open socket, and textual representation of the return values are examples of the annotation blocks. Some may argue that annotations are another form of feature engineering. However, we barely utilize the arguments and return values of the invocations to build a compound representation of the target behavior.

Context: Most malicious applications remain idle until the new device event occurred, like receiving SMS or activating the WiFi. Hence, we declared context blocks for all possible stimulation events apart from specific conditions like high cache activity or battery consumption. This contextual representation of the behaviors characterizes the malware's activities and allows malware analysts to dissect the sample profoundly.

Auxiliary: Although TRAPDROID is not primarily focused on static analysis of malware, it can extract beneficial auxiliary information from samples. In this research, the auxiliary information is out-of-scope and not integrated into our detection system. It barely provides supportive results to dynamic analysis.

Network: TRAPDROID can transparently sniff the network connections of the target application. Therefore, we can access the raw and parsed PCAP file containing network traffic while declaring a new network block in the UPL scripts.

The below code excerpt demonstrates the block definitions of the UPL scripts. For the sake of simplicity, we have eliminated redundant blocks in the whole script developed for the TRAPDROID.

```
annotation invoke_not_permitted {
  where := current.result == OPERATION_NOT_PERMITTED
  union := base_annotations
}

annotation ext_apk {
  where := current.args.extension == "apk"
  union := file_annotations
}

annotation tcp_communication {
  where :=  current.args.type == SocketType.TCP
  union := socket_annotations
}

behavior thread_create {
  where := current.target_type == TargetType.THREAD
  meta := "Thread[{current.tgid}]"
  union := base_annotations
}

behavior api_device_policy_manager {
  where := args.call == "android.app.admin.IDevicePolicyManager"
  meta := "DevicePolicyManager:{args.code}"
}

context high_stime_activity {
  where := sum(current.stime.list()) > sum(current.utime.list())
  follow := sum(current.stime.list()) > sum(current.utime.list())
}

auxiliary duplicated_permissions {
  where := len(set(app.permissions)) != len(app.permissions)
}

network high_entropy_domain_name {
  where := any([for pkt in net.dns if entro(pkt.domain) > 80.0])
}
```

Listing 3.3 Example UPL script demonstrates block definitions.

3.6 Unified Behavior Format (UBF)

UBF (Unified Behavior Format) is a simple yet powerful behavior representation format developed to model the application's dynamic activities under inspection. TRAPDROID converts pre-processed logs into UBF by processing them using a custom DSL (Domain-specific language) called UPL (UBF Processing Language). Let $E = \{e_1, e_2, ..., e_n\}$ express the observable events and their respective fields like system call arguments, API parameters, process details collected from the device and $S = \{s_1, s_2, ..., s_n\}$ indicate the system-wide measurable information like cache usage, kernel structure values, battery consumption, and memory access patterns where n is the total number of activities. The Analysis Engine (AE) is responsible for combining and transforming the S and E into a set of UBF events. Figure 3.7 depicts the whole process.

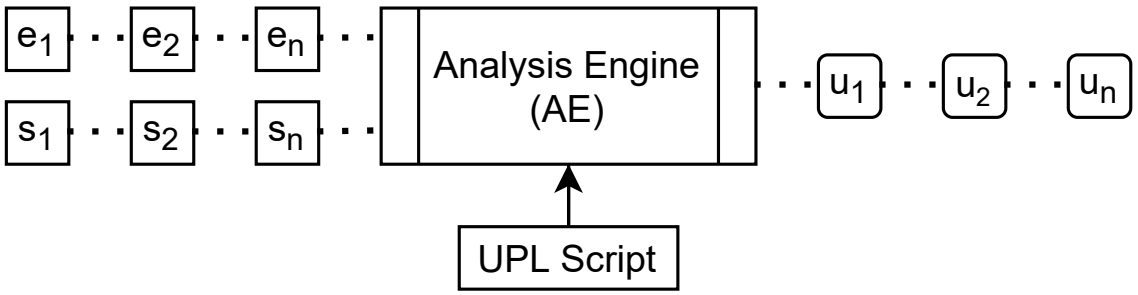

Figure 3.7 Transformation process of the raw logs into UBF.

In the UBF, we abstract the system calls, binder transactions, and broadcast actions to model the behavior of applications during our analysis. Besides abstraction, we also map the system calls that operate similarly into the same behavior. For example, if a process uses an `open` or `openat` system call to open a file, we map this behavior into `file_open`.

The kernel returns a handler, representing operating system objects like file, socket, thread, or process upon invoking the system call. For instance, `open` system call returns with a file descriptor. The Analysis Engine (AE) is capable of resolving dependencies between system calls by tracking these handlers. In order to illustrate this as an example, the process first opens a file, duplicates fd using `dup3`, invokes `ioctl` call, and closes it. We can represent the flow between these calls using the file descriptor object, which is returned from the first `open` system call and duplicated with `dup3`. Therefore, we can make better distinctions between the calls, like closing the socket or closing the file.

We also take advantage of system call arguments to produce a meaningful representation of the application's behavior. For example, if the process opens a file with the `O_RDONLY` flag, it becomes clear that the process only wants to read the file and will not be expected to alter or delete it. On the contrary, if the process opens the same file with `O_APPEND` flag, this means that the process wants to append data into the file. These represent different intents. This small but essential detail allows us to learn more information about a system call, and it enables us to make better reasoning about the process than other techniques in the literature.

The Analysis Engine maps the system call arguments into meaningful annotations specified in the UPL like `tcp_communication`, `http_communication`, `read_only`, `write_only`, etc. Each UBF block might contain zero or more annotations to enhance the activity. Table 3.4 represents the some of the sample blocks which can be activated in the running UPL script. In this research, we utilize the behavior and annotation blocks of the UBF. Other blocks are out-of-scope and need further research to find proper methods to mount that blocks into the current classification schemes.

Table 3.4 Sample blocks from defined in UBF fields.

Behavior	file_open,file_access_check,file_chmod,file_chown, file_remove,file_mkdir,file_list,file_rename,socket_create, socket_connect, process_create,thread_create,process_exc
Annotation	folder_sdcard,folder_system,folder_dev,ext_apk,ext_txt, ext_xml,ext_archive,ext_so,ext_db,access_read, open_read_only, invoke_success,invoke_not_permitted
Context	sms_received,wifi_state_changed,boot_completed, screen_on,package_installed,user_present,rebooted
Auxiliary	duplicated_permissions,vt_match,malformed_cert
Network	high_entropy_domain_name,udp_traffic

In addition to system call arguments, we also transform the return values of the system calls into meaningful text representations. The return values of the system calls depend on the type of the system call. For example, the kernel returns FD that represents the opened file; whereas, it returns PID representing the child process. In addition to that, we also use the error codes from the return values of the system calls. If a process attempts to open a file that does not exist in the file system, the kernel returns `ENOENT` error code to the user. The analysis engine transforms that error code to `no_such_file_or_directory` annotation and adds it into the active application's UBF stream. These annotations lead us to determine whether

the process tries to open a file without required permission, exploits some vulnerabilities in the operating system, or gives us signs of the behavior of applications like ransomware.

Similarly, we represent the operating system and mobile device's context information like receiving SMS, installing new applications, rebooting the device, or high cache usage in the context block of the UBF. This representation allows us to determine and bound context to activities. In this research, we selected some of the fundamental fields of the UBF. For instance, we did not include Auxiliary or Network outputs and accounting information of the target process into the detection system. Hence, further analysis is needed to bridge the semantic gap between low-level features and application-level activities.

3.7 Behavior Coverage (λ) Analysis

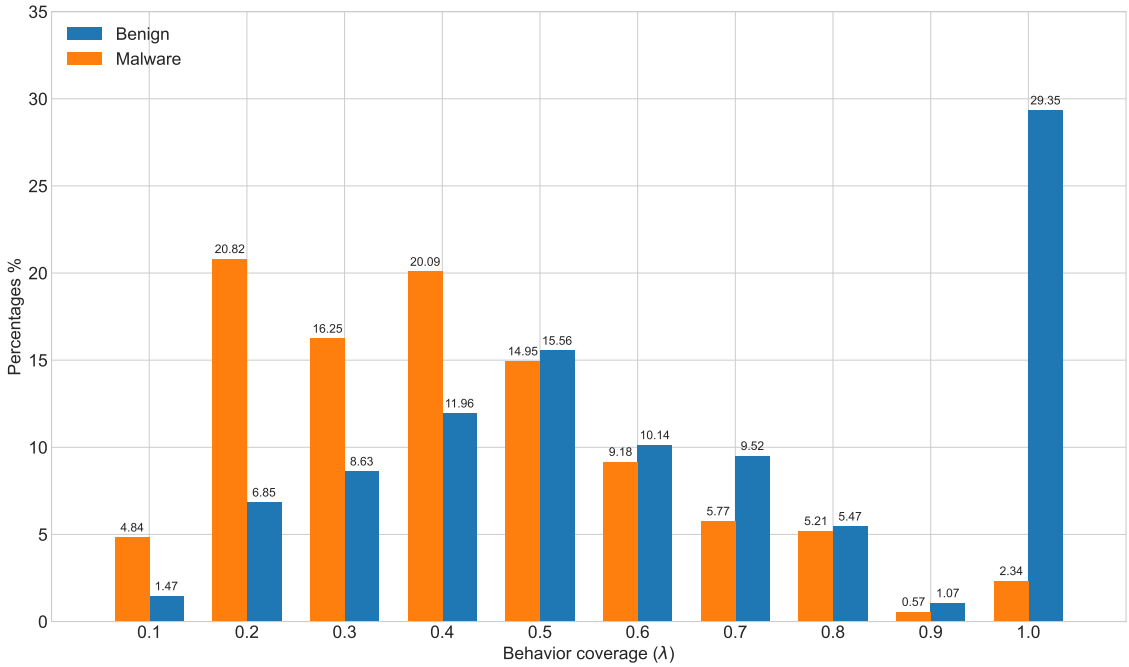

Figure 3.8 Distribution of the behavior coverage (λ).

Malware developers set necessary permissions in the manifest file to perform restricted operations like reading SMS, dialing a phone number, or accessing the internet. We utilized these permissions and dynamic behaviors to estimate the performance of the stimulation engine and reduce the noise in the dataset. Therefore, we have developed a permission-centric metric named behavior coverage (λ). This mapping is not intended to cover all possible Android permissions, and it barely represents the behavior expectations. In order to calculate the metric, the permissions defined in the APK manifest file are directly mapped to behaviors specified in the UPL blocks thanks to PScout[6] (Au et al., 2012). This mapping might be in one-to-one or one-to-many format due to the nature of the Android permission system.

[6]https://security.csl.toronto.edu/pscout/

Figure 3.8 represents the behavior coverage distribution of our dataset. One descriptive finding on distribution is that 29% of the benign samples which are not declared any permission in their manifest file produce a 1.0 behavior coverage rate. Besides, 60% of the malware samples produce less than or equal to 0.4 behavior coverage which is adequate based on our empirical findings. We deeply evaluated the effects of the coverage metric on classifier performance in Section 4.3.1, and we determined that $\lambda = 0.2$ and more is a satisfactory threshold for the detection systems.

```
behavior api_access_camera {
  where := args.call == "android.hardware.ICameraService"
  meta := "AccessCamera:{args.code}"
  depends := android.permission.CAMERA
}

behavior api_device_policy_manager {
  where := args.call == "android.app.admin.IDevicePolicyManager"
  meta := "DevicePolicyManager:{args.code}"
  depends := android.permission.BIND_DEVICE_ADMIN
}

behavior api_sms {
  where := args.call == "com.android.internal.telephony.ISms"
  meta := "SMS:{args.code}"
  depends := android.permission.RECEIVE_SMS,
             android.permission.SEND_SMS
}
```

Listing 3.4 Example UPL script represents behavior-permission mappings.

The behavior or annotation blocks in the UPL script can declare the "depends" field that builds mappings between permissions and activities. For instance, if any application sets Internet permission, the Analysis Engine expects the `connect` syscall upon `socket` syscall in the event stream. Similarly, access to camera API depends on Camera permission. Excerpt UPL code block above depicts the mappings.

4. EVALUATION

Since TRAPDROID uses Android 10 and the up-to-date kernel version, some of the samples coming from widely-known datasets have not worked as intended. Besides compatibility issues, the samples could not produce high behavior coverage because their command and control servers are already shut down. Upon successfully building a diverse dataset, we tested well-known machine learning algorithms and deep learning models using different versions of the UBF encoding such as UBF-A, UBF-P, and UBF-R on 0-day samples that are not publicly available, and 1-day samples emerged during our research. Figure 4.1 describes our evaluation steps.

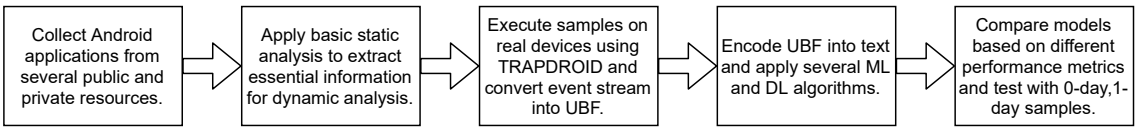

Figure 4.1 Detection flow.

The classical machine learning (ML) algorithms like SVM, Random Forest, or Gradient Boosting Trees produce outstanding performance on malware analysis. However, the success of the malware detection systems profoundly depends on the specially crafted feature vectors and authors' security background. Recent publications show us that feature engineering is time-expensive, error-prone, task-specific, and subjective to individual judgment (Qiu et al., 2020). In our research, we do not perform any feature engineering or selection to manipulate the output of algorithms. Instead, the machine learning algorithms treat Android malware detection as a traditional text classification problem.

Besides classical machine learning algorithms, we also tested our UBF format using novel deep neural computation networks. Deep learning is an excellent approach to overcoming feature-engineering obstacles and provides generalization for malware detection. Furthermore, we apply several text classification models of deep neural networks and novel approaches to improve the performance of our detection system.

Most of the proposed models are predominantly influenced by text classification approaches. Hence, Attention (Vaswani et al., 2017), Transformer (Wang et al., 2019), or similar state-of-the-art text classification techniques are also applicable to our problem domain.

4.1 Dataset

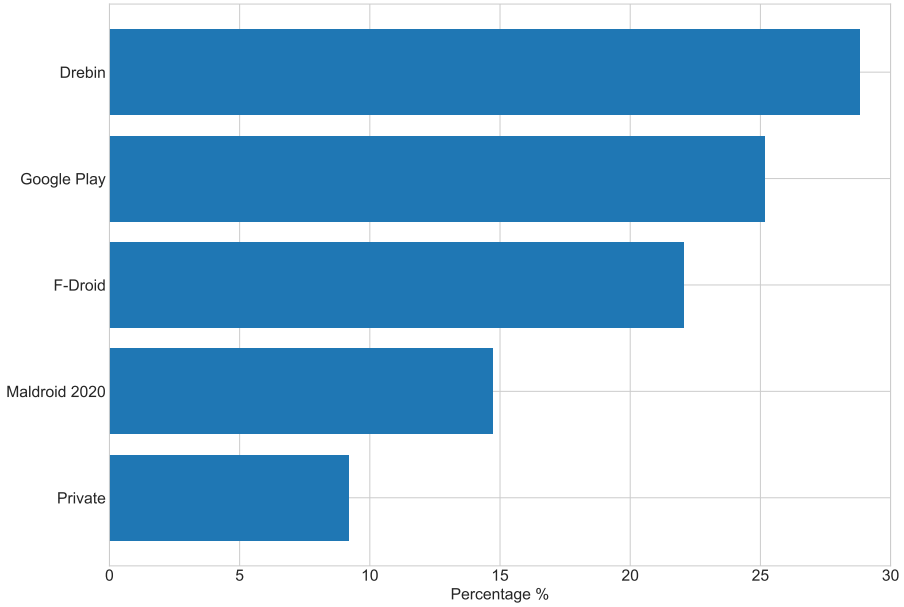

Figure 4.2 Source distribution of the our new dataset.

The Drebin dataset (Arp et al., 2014), which is collected in 2014, portrays the historical Android malware samples up to that date. Nevertheless, threat actors constantly evolve and seek new abusive functionalities in the operating system to make their strain undetectable or offensive regarding exfiltration, device management, or data acquisition methods. To discover 0-day or 1-day malware samples with high accuracy, we need to update existing datasets by feeding them with distinct malware families. Therefore, we have collected 3638 benign and 3701 malicious samples from several resources including Drebin, Maldroid (Mahdavifar et al., 2020), Google Play,[1] F-Droid,[2] and professional security researchers. Figure 4.2 depicts the source distribution of our dataset.

[1]https://play.google.com/store

[2]https://www.f-droid.org/en/packages/

In order to increase the diversification and reduce the noise in the dataset, we have evaluated the collected samples based on different criteria like defined permission distribution, behavior coverage, number of family variations, and VirusTotal[3] detections. We also removed 3.92% of all collected samples due to their dormant state or execution problems.

Virustotal We expect that malware samples have at least one AV detection and zero detection rate for benign samples. Some of the edge samples include benign AV software with high detection rate or custom zero-permission malware Zpware are also allowed.

Behavior Coverage (λ) There is a trade-off between behavior coverage and dataset size due to the nature of malware. We determined that $\lambda = 0.2$ is a reasonable threshold for the stimulation engine. Hence, our dataset contains samples that have at least revealed 20% of the whole possible activities.

Malfunction Some of the samples, mostly coming from Drebin, cannot work as intended, wherefore compatibility issues, C2 communication problem, and restricted API usage. We eliminated those automatically upon analysis completed.

Permission Distribution We do not allow some malware families or samples to dominate the entire database regarding permission distribution. The analysis engine is trying to keep that distribution always balanced using several conditions.

Category The benign samples coming from Google Play and F-Droid are carefully and uniformly selected based on their purpose. Most benign samples are widely known and installed by mobile device users for usage in daily life. Banking, game, antivirus, system, SMS, or multimedia are excerpts of the complete category list.

0-day or 1-day samples We have been collecting malware samples upon emerged in public since the beginning of 2021. In addition to 1-day samples, some threat intelligence professionals provide to us 0-day samples that are not publicly available.

[3] https://www.virustotal.com/

4.2 Metrics

TP denotes the number of true positives, FP the number of false positives, TN the number of true negatives, and FN the number of false negatives. Thus, while Recall (R) is the sensitivity metric of the detection formularized using TP and FN, Precision (P) is the relevance metric of the detection formularized using TP and FP.

The F-1 score is a compound metric that takes advantage of precision and recall. In order to develop a more robust classifier, we utilized F1-score for each class without using pre-defined weights.

MWR is a metric that calculates the malware ratio of the subset. For instance, the dataset indicates that MWR equals 0.4 and dataset size is 1000, which means that 400 malware samples available in the subset with 600 benign samples. Moreover, MWR=0.5 indicates that the dataset is balanced. For the balanced datasets, accuracy could be used for performance metrics rather than the F-1 score. However, the F-1 metric is crucial for imbalanced datasets.

Table 4.1 The evaluation metrics.

Metric	**Explanation**
True Positive (TP)	# of correctly identified malware samples
False Positive (FP)	# of incorrectly identified benign samples
True Negative (TN)	# of correctly identified benign samples
False Negative (FN)	# of incorrectly identified malware samples
Precision (P)	$= \frac{TP}{TP+FP}$
Recall (R)	$= \frac{TP}{TP+FN}$
Accuracy (ACC)	$= \frac{TP+TN}{TP+FP+TN+FN}$
F-Score (F-1)	$= 2\frac{P \times R}{P+R}$
Malware Ratio (MWR)	$= \frac{\text{\# of malware samples}}{\text{\# samples in dataset}}$
Behavior Coverage (λ)	$= \frac{\text{\# of occurred behaviors linked with regarding permission}}{\text{\# of defined permissions}}$

4.3 TF-IDF

As we described in Section 3.6, we convert the event stream into UBF (Unified Behavior Format), a novel representation format developed for modeling dynamic activities of the sample under the analysis. Then, we encode some selected fields of the UBF to vector format, which is the list of tokenized strings.

Let $D = \{a_1, a_2, ..., a_N\}$ denotes the dataset which contains N distinct application samples and $a = \{u_1, u_2, ..., u_n\}$, where n is the total number of UBF event happened under the analysis of sample application a. We define the TF and IDF values of the UBF event $u_i \in a$ as follows:

$$\text{TF}_i = \frac{f_i}{n} \qquad\qquad \text{IDF}_i = log(\frac{N}{df_i})$$

where f_i is the frequency of u_i appearing in application a_i; and df_i is the number of all applications contain the u_i. Based on the equations, we can define the TF-IDF value of the UBF event u_i as follows:

$$\text{TF-IDF}_i = \text{TF}_i \times \text{IDF}_i \tag{4.1}$$

In order to keep the local order information of the UBF events, we apply the n-grams technique before building the TF-IDF vector. While n-grams increase the vector size excessively, it encodes the sequential combinations of the behaviors. $n = 3$ provides the best performance when considering the computational requirements, time-costs, vector size, and evaluation metrics. According to Table 4.2, we can infer that most of the crucial features selected by the classifiers are n-grams of the tokens. Despite these promising results, n-grams are not providing the descriptive output of the annotations and context information of the UBF. We overcome these limitations using the separate word embedding matrices for each input in the deep neural network.

Table 4.2 Some of the selected best features.

UBF-A	**UBF-P**
no_such_file, read_write	iactivitymanager:184
iservicemanager:1, itelephony:145	igraphicsstats:1
ipackagemanager:13, iwindowmanager:37	file_remove, file_open, file_chmod
folder_data_user, file_chmod, success	itelephony:145
read_write, create_file, exclude_file	iservicemanager:1, itelephony:145
itelephony:123	itelephony:123
iphonesubinfo:13	file_open, file_remove
iservicemanager:1, isub:29	iphonesubinfo:13
iservicemanager:1, iphonesubinfo:13	iservicemanager:1, itelephony:123
igraphicsstats:1, iwindowsession:1	iservicemanager:1, iphonesubinfo:13
folder_data_user, file_remove, success	iactivitymanager:184, igraphicsstats:1

We developed three different encoding formats for the tokenization of the UBF: UBF-P, UBF-R, and UBF-A. While UBF-P utilizes the Behavior name field of the UBF and has the same characteristics as the traditional malware detection system built on top of system and API calls, UBF-R is an extended version of the UBF-P, which couples with their return value. Likewise, UBF-A is the improved variant of the UBF-P, enhanced with well-defined annotations of the events. Figure 4.3 depicts the tokenization process of the UBF event.

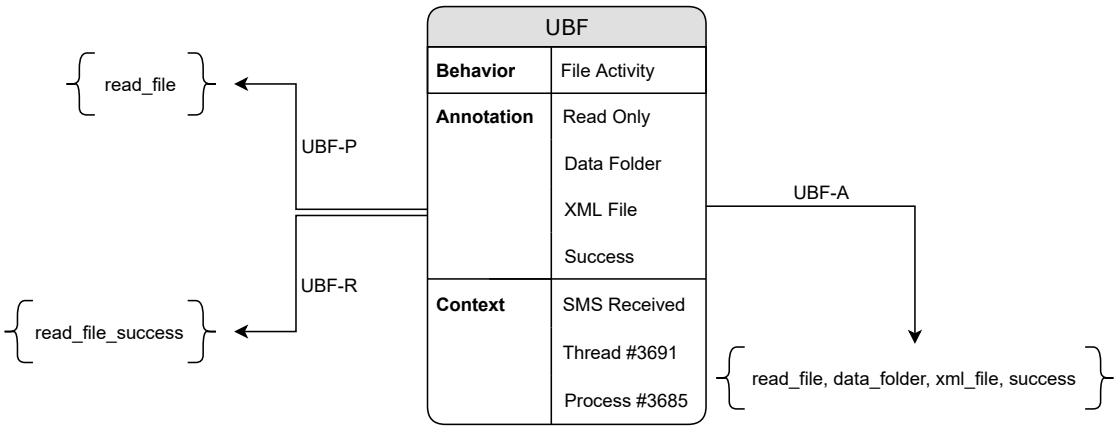

Figure 4.3 Sample representation of the tokenization process of UBF.

4.3.1 Impact of the Behavior Coverage (λ)

First of all, we tested the F-1 performance of various behavior coverage (λ) metric on selected dataset sizes (N) in with 80%, 20% train-test split rate (Prusa & Khoshgoftaar, 2016). $\lambda = 0$ means that the application under the analysis did not reveal their coded behaviors while performing stimulation. In this condition, the performance of the classifiers is reduced drastically and may cause an overfitting problem. On the contrary, $\lambda = 1.0$ refers to successful stimulation, meaning that the application showed all its expected behaviors. As we can see in Figure 4.4, there is a strong correlation between behavior coverage and performance of the classifier until they reach a certain dataset size. Besides behavior coverage, dataset size plays an indispensable role in the system's performance. It should be noted that behavior coverage needs to be selected based on the expectations and resource limitations.

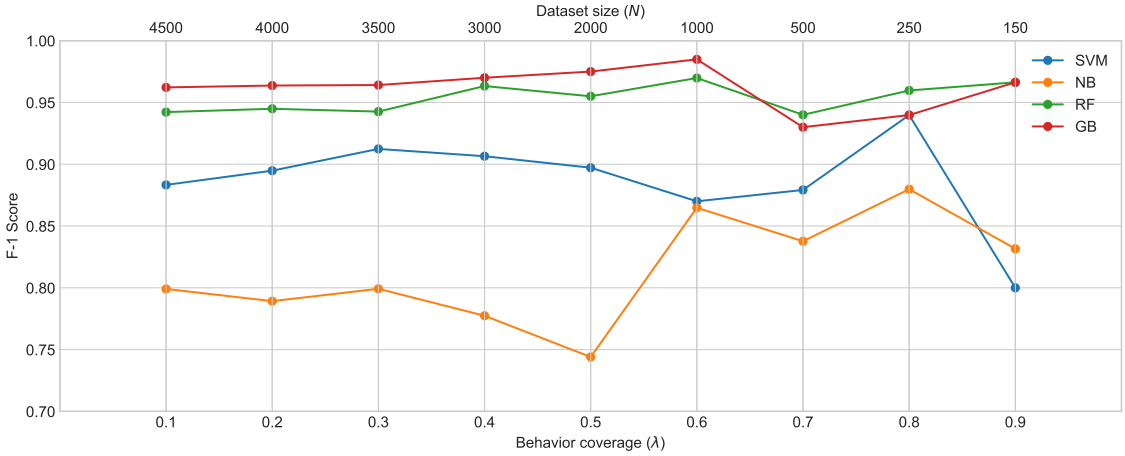

Figure 4.4 Behavior coverage (λ) performance over balanced (MWR=0.5) dataset size.

Although the performance of the classifiers continues to increase as the behavior coverage increases, we determined that $\lambda = 0.2$ and more is a reasonable threshold to reduce the noise in the dataset and avoid the overfitting problem because dataset size (N) is limited. Thus, according to these findings, we can infer that high behavior coverage enables detection systems to provide their best performance in large dataset sizes.

4.3.2 Baseline Performance

We tested baseline performance of the SVM (Support Vector Machines), GB (Gradient Boosted Tree), and RF (Random Forest) algorithms on the 3000 samples using $\lambda = 0.2$ without optimization in order to figure out the effects of the MWR and different tokenization techniques over performance. As shown in Table 4.3, UBF-P reaches the best F-1 value which is 97.10% with GB classifier where MWR=0.4 and its accuracy reaches 97.50% where MWR=0.7.

Table 4.3 Baseline performance of the UBF-P with TF-IDF.

Classifier	**MWR**	**P**	**R**	**F-1**	**ACC**
SVM	0.3	0.9393	0.8991	0.9153	0.9283
	0.4	0.9264	0.9168	0.9207	0.9233
	0.5	0.9094	0.9094	0.9094	0.9100
	0.6	0.9209	0.9145	0.9175	0.9233
	0.7	0.9143	0.9066	0.9103	0.9283
RF	0.3	0.9700	0.9407	0.9534	0.9600
	0.4	0.9632	0.9554	0.9587	0.9600
	0.5	0.9541	0.9557	0.9548	0.9550
	0.6	0.9580	0.9564	0.9572	0.9600
	0.7	0.9513	0.9360	0.9433	0.9550
GB	0.3	**0.9722**	0.9635	0.9677	0.9717
	0.4	0.9712	0.9707	**0.9710**	0.9717
	0.5	0.9696	0.9701	0.9698	0.9700
	0.6	0.9615	0.9640	0.9627	0.9650
	0.7	0.9667	**0.9717**	0.9692	**0.9750**

UBF-R outperforms the UBF-P with 97.53% F-1 score and 98.00% accuracy when dataset is highly imbalanced (MWR=0.7). Most of the time, UBF-R is more solid than UBF-P in terms of overall performance.

Table 4.4 Baseline performance of the UBF-R with TF-IDF.

Classifier	**MWR**	**P**	**R**	**F-1**	**ACC**
SVM	0.3	0.9545	0.9218	0.9356	0.9450
	0.4	0.9195	0.9130	0.9158	0.9183
	0.5	0.9181	0.9188	0.9183	0.9183
	0.6	0.9237	0.9230	0.9233	0.9283
	0.7	0.9109	0.9168	0.9138	0.9300
RF	0.3	**0.9774**	0.9520	0.9633	0.9683
	0.4	0.9659	0.9593	0.9622	0.9633
	0.5	0.9504	0.9495	0.9499	0.9500
	0.6	0.9503	0.9461	0.9481	0.9517
	0.7	0.9375	0.9158	0.9259	0.9417
GB	0.3	0.9748	0.9685	0.9715	0.9750
	0.4	0.9757	0.9731	0.9743	0.9750
	0.5	0.9752	0.9747	0.9749	0.9750
	0.6	0.9672	0.9688	0.9680	0.9700
	0.7	0.9736	**0.9770**	**0.9753**	**0.9800**

From the data in Table 4.5, it is apparent that the UBF-A outperforms the other proposed tokenization schemes, UBF-R and UBF-P, with 97.78% F-1 score. Moreover, a closer inspection of the table shows GB (Gradient Boosted Trees) classification algorithms considerably better than other tested algorithms.

Table 4.5 Baseline performance of the UBF-A with TF-IDF.

Classifier	**MWR**	**P**	**R**	**F-1**	**ACC**
SVM	0.3	0.9191	0.8942	0.9050	0.9183
	0.4	0.8779	0.8720	0.8745	0.8783
	0.5	0.8914	0.8917	0.8915	0.8917
	0.6	0.9038	0.9038	0.9038	0.9100
	0.7	0.8921	0.8877	0.8899	0.9117
RF	0.3	0.9758	0.9533	0.9634	0.9683
	0.4	0.9583	0.9530	0.9554	0.9567
	0.5	0.9534	0.9531	0.9532	0.9533
	0.6	0.9517	0.9412	0.9460	0.9500
	0.7	0.9437	0.9182	0.9298	0.9450
GB	0.3	0.9773	0.9698	0.9734	0.9767
	0.4	**0.9786**	0.9770	**0.9778**	0.9783
	0.5	0.9750	0.9749	0.9749	0.9750
	0.6	0.9693	0.9702	0.9698	0.9717
	0.7	0.9764	**0.9782**	0.9773	**0.9817**

4.3.3 Impact of the Malware Ratio (MWR)

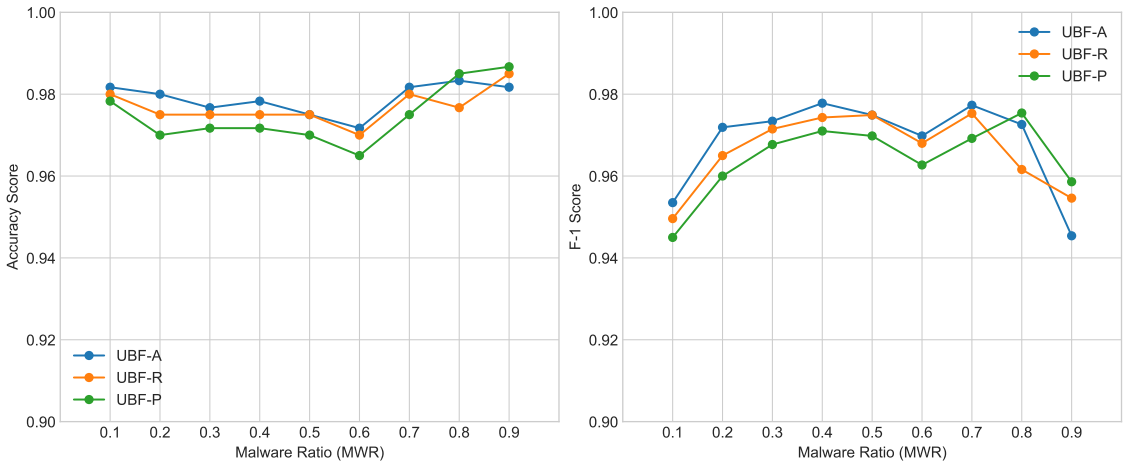

Figure 4.5 Performance of the datasets with various Malware Ratios (MWRs).

Malware ratio (MWR) is a crucial metric that provides information on the imbalance of the dataset. Moreover, it directly affects the classifier performance and needs to be determined precisely. From the data ($\lambda = 0.2$) in Figure 4.5, it is apparent that extremely imbalanced datasets produce high-accuracy scores. On the other hand, the F-1 score of the classifier rising as the malware ratio approaches 0.5. When the malware ratio equals 0.4,0.5 or 0.7, the classifier reaches top scores in terms of accuracy and F-1 scores. Therefore, MWR can be adjusted to develop a successful detection system for high detection or low false-positive rate.

4.3.4 Optimization

After carefully analyzing the output of the baseline classifiers, we decided to optimize the parameters of the algorithms using a tree-based pipeline optimization (Olson et al., 2016). Moreover, we observed that feature selection based on SVM feature importances slightly increases the system's performance and reduces the size of the input vector of the classification algorithm. As a result, UBF-P, UBF-R, and UBF-A attain F-1 scores of 97.75%, 97.92%, and 98.08%, respectively, when using the balanced dataset (MWR=0.5). In addition to the F-1 score, UBF-A attains 98.08% accuracy with relatively low false-positive rate. Table 4.6 depicts the output of the classifiers that reached the best performance with optimized parameters and feature selection on 6000 randomly selected samples.

Table 4.6 Optimized performance of TF-IDF with MWR=0.5 and $\lambda = 0.2$.

Tokenization	**Classifier**	**P**	**R**	**F-1**	**ACC**
UBF-P	GB	0.9775	0.9777	0.9775	0.9775
UBF-R	GB	0.9791	0.9793	0.9792	0.9792
UBF-A	GB	0.9808	0.9809	0.9808	0.9808

4.4 Deep Learning Models

Deep learning is a powerful approach for malware detection and researchers are constantly developing new models to overcome the limitations of traditional machine learning algorithms. The Convolutional Neural Network (CNN) is another type of deep learning model developed for image processing problems, and its performance mainly depends on the convolution function and sliding filter. It is a widely accepted approach to solve text classification and sequence processing problems. We employed a simple text-classification CNN model to the malware detection domain and improved it by attaching different combinations of embedding layers. In this section, we first present our novel CNN models containing single and multiple embedding layers. Then, we analyze the performance of our networks on malware detection.

4.4.1 CNN with Single Embedding Layer

One-hot-encoding is a method to produce a sparse binary vector that allows the representation of texts in vector space. It is widely accepted and used in text classification problems. However, it has several disadvantages, like producing gigantic vectors and inadequately reflecting semantic distances of the objects. In this research, we employed an embedding layer for compact representation, which includes semantic information. Formally, an embedding is defined by a dense matrix $W \in \mathbb{R}^{d_v \times d_w}$ with d_v the size of the vocabulary and d_w the dimension of the embedding, so that $d_w \ll d_v$. The size of the vocabulary d_v highly depends on the sequence length n.

Table 4.7 Proposed CNN architecture.

Layer	**Output Shape**
Embedding	(None, n, d_w)
Conv1D	(None, n, 50)
GlobalMaxPooling1D	(None, 50)
Dense(ReLU)	(None, 50)
Dense(ReLU)	(None, 100)
Dense(ReLU)	(None, 50)
Dense(Sigmoid)	(None, 1)

A convolutional layer in deep neural networks encodes the patterns of local structure in the input sequences. We connected a pooling layer following the convolutional layer to perform maximum pooling operation through enhancing the sequences. Moreover, we added a Dense layer with a stall regulation rate of 0.6 (probability of 0.6 that a given element is dropped during training) using a rectification function (ReLu) as the activation function. Table 4.7 represents the designed and implemented network architecture, where n refers to sequence length and d_w refers to the embedding layer's output dimension.

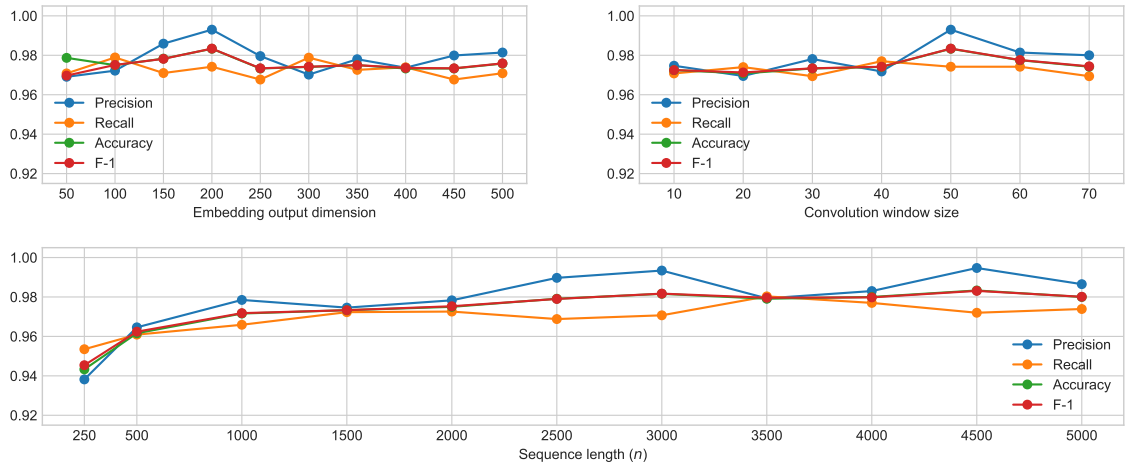

Figure 4.6 Performance comparison of the proposed CNN architecture with different parameters.

We tested our proposed network on the same 6000 samples with MWR=0.5 and $\lambda = 0.2$. Figure 4.6 represents the results obtained from the preliminary analysis of our neural network architecture with different parameters. The CNN network achieves best performance when embedding output dimension $d_w = 200$ and sequence length $n = 4500$. Likewise, the convolutional layer (Conv1D) directly affects the performance, and we determined the 50 and 5 for window and kernel size, respectively.

Table 4.8 shows the results of experiments performed to evaluate the performance of the proposed neural model with different combinations of the inner architecture of the dense layers. With the $[50, 100, 50]$ combination, we reached a 98.31% F-1 score and an 98.33% accuracy.

Table 4.8 Comparison of the different hidden layer combinations.

# of layers	**# of neurons**	**P**	**R**	**F-1**	**ACC**
1	50	0.9829	0.9656	0.9740	0.9741
1	100	0.9799	0.9728	0.9761	0.9758
1	200	0.9894	0.9639	0.9763	0.9766
1	250	0.9797	0.9691	0.9742	0.9741
1	300	0.9862	0.9691	0.9774	0.9775
2	50, 50	0.9798	0.9741	0.9766	0.9766
2	100, 100	0.9846	0.9701	0.9771	0.9775
2	200, 200	0.9766	0.9741	0.9751	0.9750
2	250, 250	0.9717	0.9789	0.9750	0.9750
2	300, 300	0.9586	**0.9852**	0.9714	0.9708
3	50, 50, 50	0.9850	0.9643	0.9743	0.9741
3	50, 100, 50	**0.9947**	0.9720	**0.9831**	**0.9833**
3	100, 100, 100	0.9896	0.9673	0.9782	0.9783
3	100, 50, 100	0.9780	0.9754	0.9764	0.9766
3	100, 200, 100	0.9845	0.9689	0.9764	0.9766
3	200, 200, 200	0.9695	0.9819	0.9753	0.9750
3	200, 300, 200	0.9748	0.9706	0.9724	0.9725
3	300, 300, 300	0.9815	0.9725	0.9767	0.9766
4	50, 50, 50, 50	0.9766	0.9673	0.9717	0.9716
4	50, 100, 100, 50	0.9798	0.9754	0.9775	0.9775
4	100, 100, 100, 100	0.9754	0.9775	0.9761	0.9758
4	200, 200, 200, 200	0.9897	0.9689	0.9791	0.9791

4.4.2 CNN with Multiple Embedding Layers

In order to include the annotations into the deep-neural network, we employed multiple embedding layers for each input sequence. For the sake of simplicity, we selected two input sequences: Behavior and Annotations. While the Behavior sequence contains the tokenized UBF-R stream of events, the Annotations sequence consisted of a list of annotations whose size is limited to 16. Figure 4.7 depicts the vectorization process of the UBF.

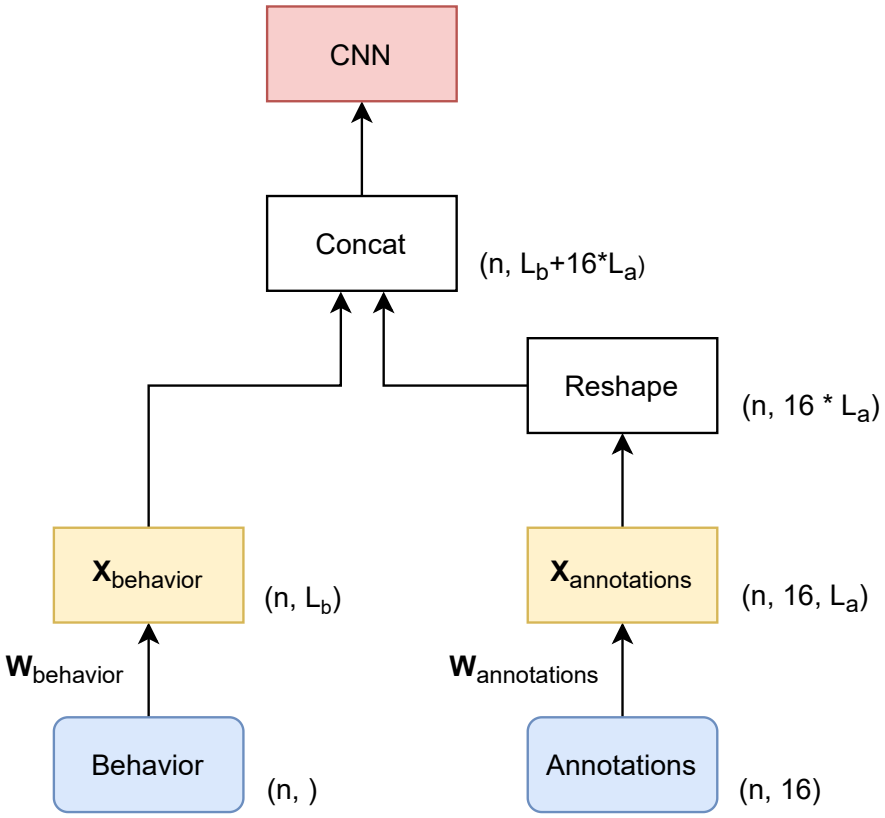

Figure 4.7 Simplified representation of the vectorization process of the UBF.

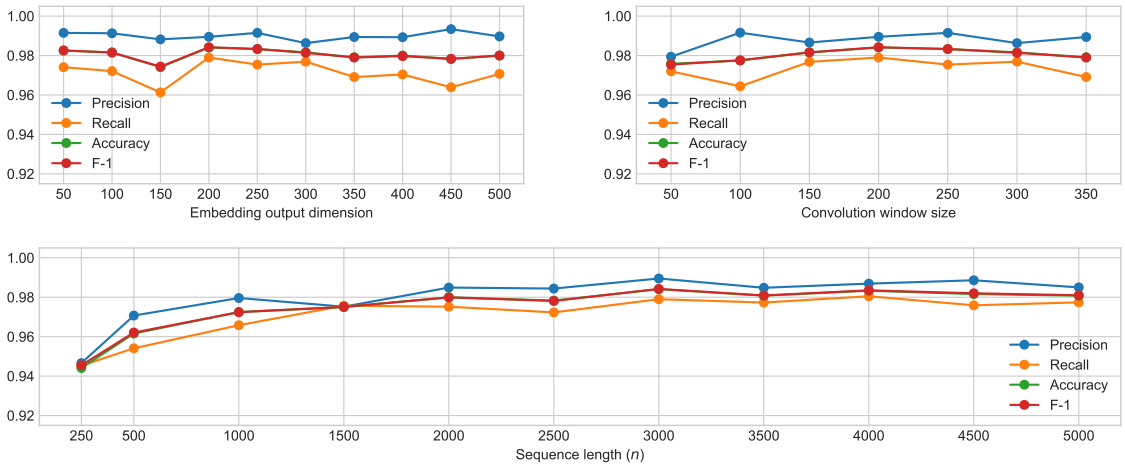

Figure 4.8 Performance comparison of the proposed CNN network contains multiple embedding layers.

Upon the vectorization process, we tested our proposed network on the same 6000 samples with MWR=0.5 and $\lambda = 0.2$. Figure 4.8 represents the results obtained from the extensive analysis of our neural network architecture contains multiple embedding layers. The CNN network achieves the best performance when embedding output dimension $d_w = 200$ and sequence length $n = 3000$. Likewise, the convolutional layer (Conv1D) directly affects the performance, and we determined the 200 for window size.

Table 4.9 shows the results of experiments performed to evaluate the performance of the proposed neural model with different combinations of the inner architecture of the dense layers. With the [100, 100] combination, we reached a 98.42% F-1 score and an 98.41% accuracy.

Table 4.9 Performance of the CNN model contains multiple embedding layers with various configurations.

# of layers	**# of neurons**	**P**	**R**	**F-1**	**ACC**
1	50	0.9830	0.9740	0.9783	0.9783
1	100	0.9829	0.9721	0.9772	0.9775
1	200	0.9846	0.9804	0.9823	0.9825
1	250	0.9828	0.9725	0.9774	0.9775
1	300	0.9782	**0.9820**	0.9800	0.9800
2	50, 50	0.9829	0.9721	0.9772	0.9775
2	100, 100	**0.9895**	0.9790	**0.9842**	**0.9841**
2	200, 200	0.9880	0.9743	0.9810	0.9808
2	250, 250	0.9814	0.9736	0.9774	0.9775
2	300, 300	0.9702	0.9868	0.9783	0.9783
3	50, 50, 50	0.9878	0.9737	0.9806	0.9808
3	50, 100, 50	0.9880	0.9735	0.9805	0.9808
3	100, 100, 100	0.9865	0.9739	0.9800	0.9800
3	100, 50, 100	0.9845	0.9724	0.9783	0.9783
3	100, 200, 100	0.9895	0.9688	0.9789	0.9791
3	200, 200, 200	0.9750	0.9803	0.9774	0.9775
3	200, 300, 200	0.9814	0.9736	0.9774	0.9775
3	300, 300, 300	0.9880	0.9770	0.9823	0.9825
4	50, 50, 50, 50	0.9879	0.9654	0.9763	0.9766
4	50, 100, 100, 50	0.9847	0.9719	0.9780	0.9783
4	100, 100, 100, 100	0.9878	0.9687	0.9778	0.9783
4	200, 200, 200, 200	0.9784	0.9775	0.9777	0.9775

5. RESULTS AND DISCUSSION

This chapter includes the comparison results of our framework with existing research. We present 0-day and 1-day malware detection performances of different methodologies. Moreover, several dynamic features, advanced use-cases, and remarkable observations are also discussed in this section. While we analyze our system's detection performance, we additionally inspected the inner structures of the most widespread malware variants. Thus, we proposed a protection mechanism to limit their capabilities in mobile devices.

5.1 Comparison with Other Approaches

In this section, we present our results from the TRAPDROID and compare them with other state-of-the-art research. First, we discuss the performance of the classifiers. Subsequently, we compare the methodologies of both approaches. This is followed by the final discussion of platform features.

5.1.1 Platform Capabilities

Table 5.1 illustrates some of the main capabilities of the available platforms. TRAPDROID seems more capable than others to collect information from multiple resources like system calls, IPC communications, PMU, battery, or kernel structures. In contrast to capabilities, the most significant limitation lies in the fact that TRAPDROID needs more resources to deploy and maintain the system. Therefore, the tradeoff between resource and detection performance needs to be carefully considered.

Table 5.1 Comparison of the platform capabilities.

Name	Environment	System Calls	IPC	ptrace	PMU	Battery	Kernel Structures
BareDroid (Mutti et al., 2015)	Bare-metal	✗	✗	✗	✗	✗	✗
Crowdroid (Burguera et al., 2011)	Emulator	✓	✗	✓	✗	✗	✗
DroidTrace (Zheng et al., 2014)	Emulator	✓	✗	✓	✗	✗	✗
CopperDroid (Tam et al., 2015)	Emulator	✓	✓	✗	✗	✗	✗
DroidScope (Yan & Yin, 2012)	Emulator	✓	✓	✗	✗	✗	✗
DroidBox (Chaurasia, 2015)	Emulator	✗	✓	✗	✗	✗	✗
Andrubis (Weichselbaum et al., 2014)	Emulator	✓	✓	✗	✗	✗	✗
TRAPDROID	Bare-metal	✓	✓	✗	✓	✓	✓

5.1.2 Detection Performance

Table 5.2 compares the related works on various configurations and dataset sizes. What is interesting about the data in this table is that our proposed models and tokenization schemes outperform most of the works. Overall, these results indicate that malware detection in a bare-metal environment with realistic malware samples produces high performance on detection. In addition, it is tough to compare approaches due to the lack of information on the dataset. To overcome this drawback, we made our dataset publicly available for other researchers. Other researchers can obtain the our dataset and analyze the their detection system's performance on the same samples. They can obtain our dataset via request and analyze their detection system's performance on the same samples.

Table 5.2 Performance comparison of the selected works.

Name	Dataset Size (N)	MWR	P	R	F-1	ACC
Hou et al. (2016)	3000	0.5	0.9396	0.9336	0.9368	0.9368
Xiao et al. (2019)	7103	0.502	0.9126	0.9663	N/A	0.9367
Abderrahmane et al. (2019)	12750	0.807	0.9410	0.9780	0.9600	0.9330
Canfora et al. (2015)	2000	0.5	N/A	N/A	N/A	0.9700
Surendran et al. (2020)	2750	0.41	0.9490	N/A	0.9410	0.9540
Vinayakumar et al. (2018)	558	0.5	0.9370	0.9870	0.9610	0.9390
Karbab et al. (2016)	8639	0.395	0.9400	0.7800	0.8500	N/A
Yeh et al. (2016)	32000	0.5	N/A	N/A	N/A	0.9312
UBF-P with RF (Baseline)	4000	0.5	0.9678	0.9740	0.9707	0.9725
UBF-R with GB (Baseline)	4000	0.5	0.9720	0.9777	0.9762	0.9747
UBF-A with GB (Baseline)	4000	0.5	0.9773	0.9799	0.9786	0.9800
UBF-P with GB (Optimized)	6000	0.5	0.9775	0.9777	0.9775	0.9775
UBF-R with GB (Optimized)	6000	0.5	0.9791	0.9793	0.9792	0.9792
UBF-A with GB (Optimized)	6000	0.5	0.9808	0.9809	0.9808	0.9808
CNN (Single Embedding)	6000	0.5	0.9947	0.9720	0.9831	0.9833
CNN (Multiple Embedding)	6000	0.5	0.9895	0.9790	0.9842	0.9841

A greater focus on text classification techniques and models could produce interesting findings that account more effectively for malware detection. In this thesis, we employ TF-IDF and Embedding techniques to encode our UBF streams. Although our novel approach provides promising results, this research has thrown up many questions in need of further investigation into similarities between textual data and behavioral sequences. More work will need to be done to determine the approaches built on top of Attention, Transformer, skip-grams, or similar state-of-the-art text classification techniques. Moreover, pre-calculated embedding layers with an adequate number of samples can be distributed to boost the resource-limited detection systems. Besides, further research might explore how to bridge the semantic gap between applications and low-level observable information. In this research, we did not include that sort of data in the detection system for the sake of simplicity.

5.1.3 Environment and Data Collection Methodology

Most of the detection platforms utilize the Emulator and `ptrace` system-call-based approach. However, evasive malware can quickly identify that sort of environments and abort its execution. The most successful detection system must be highly transparent to the application under inspection. In order to increase the transparency of the detection system, we developed a Linux kernel module that allows us to collect the required information for malware detection. Furthermore, one of the essential parts of the detection system is stimulation. We employed a UI coverage-based stimulation approach to reveal more hidden behavior rather than a randomness-based approach. The evidence from this study suggests that effective stimulation techniques improve the detection as well. Table 5.3 illustrates the differences between TRAPDROID and other widely known researches in terms of methodology.

Table 5.3 Environment and data collection methodology comparison.

Name	**Environment**	**Data Collection**	**Stimulation**
Hou et al. (2016)	Emulator	strace (ptrace)	Custom
Xiao et al. (2019)	Bare-metal	strace (ptrace)	Monkey
Abderrahmane et al. (2019)	Emulator	strace (ptrace)	Monkey
Canfora et al. (2015)	Bare-metal	N/A	Monkey
Surendran et al. (2020)	Emulator	strace (ptrace)	Monkey
Vinayakumar et al. (2018)	Emulator	STREAM	Monkey
Karbab et al. (2016)	Emulator	DroidBox	Monkey
Yeh et al. (2016)	Emulator	DroidBox	Monkey
TRAPDROID	Bare-metal	Kernel-level	Improved Droidbot

5.2 0-day and 1-day Malware Detection

While 0-day malware refers to a sample found in the wild as undetected or previously unseen, 1-day malware is a fresh sample that emerges in a relatively short time. We tested our system's performance on carefully collected 0-day and 1-day samples, which can be seen in Table 5.4.

Table 5.4 Selected 0-day and 1-day samples.

Hash	Type	Variant	VirusTotal Status
e4e4c5e3b3b147910d44bbe7bc3499f6	0-day	Alien	N/A (22.05.2021)
895db415cf5e8facd0dbce2c737282e1	0-day	Alien	N/A (22.05.2021)
d073164e36b1044633f30daab18ed2ef	0-day	Alien	N/A (22.05.2021)
8ad1cbb3c7e9c9b673e5c016456e66cd	1-day	Toddler	25/64 (18.05.2021)
6c430813edb87df5f92d2b55611d6b9b	1-day	Toddler	25/63 (20.05.2021)
1a352c997a2ac7c8a18aee5c581674a7	0-day	Anubis	N/A (22.05.2021)
47154b064d9773afce6ca189d49650af	0-day	Anubis	N/A (22.05.2021)
c804f7bf78bae2e34c0a080677a16298	1-day	Flubot	7/62 (20.05.2021)
194b876c5cc689ef9e77b64d92869a10	1-day	Flubot	21/63 (13.05.2021)
c407853771c163c6b1110b5630f36ee4	0-day	Hydra	N/A (22.05.2021)
95fe97f1cdc00405518a853313a00472	0-day	Hydra	N/A (22.05.2021)
31fca10d265d00e66bf116c5c2408484	0-day	Medusa	N/A (22.05.2021)
3253e2c462bcd739a58e973448a3482c	0-day	Hydra	N/A (22.05.2021)
72d874162e112fba1c4294d372c5928a	0-day	Hydra	N/A (22.05.2021)
f5f28fda870b1cfe5fecb8ca9354dea5	0-day	Hydra	N/A (22.05.2021)

Flubot and Toddler are relatively new banking malware variants that utilize advanced persistence techniques and smishing. They both emerged during the research at the beginning of May 2021. PRODAFT,[1] a reputable cyber threat intelligence company, employed TRAPDROID to detect previously unseen malware samples and variants crawled from Darknet.[2] Nevertheless, our detection system identified most of the never-before-seen samples without re-training. This lead to the naming of the Flubot[3] as the sample was not categorized before.

[1] https://www.prodaft.com

[2] https://en.wikipedia.org/wiki/Darknet

[3] https://www.prodaft.com/m/reports/FluBot_4.pdf

One possible explanation might be that these variants heavily depend on accessibility services like Alien or other sorts of banking samples. Consequently, these results support the idea that our classifier can identify new variants without seeing them before.

Moreover, we successfully identified several 0-day samples including Alien, Anubis, Hydra, Medusa, and other variants in the wild as malware but not yet detected by the Virustotal or included in other datasets. Accordingly, most of the samples are developed by the financially motivated threat actors and represent the current Android malware trends. Besides banking malware, two new high-profile APT variants were identified using our system in the field by professional malware analysts. These findings are out-of-the-scope of this research, but it confirms the 0-day detection performance of our models. We did not include the APT examples in Table 5.4 due to the avoiding disrupting ongoing investigations.

5.3 Novel Dynamic Features

While we inspected our dataset, we discovered that some of the malware we tested were actively trying to hide their presence by calling `setComponentEnabledSetting` method of `IPackageManager`[4] class. We concluded that the observation of this function within an application effects the output of our framework drastically.

As another finding, we observed that there are significant differences between `stime` and `utime` values of a given application. Malware samples have a higher `stime` duration compared to their `utime` durations. Respectively, benign samples have a longer `utime` duration than `stime`. Additionally, `nvcsw`, `nivsw` and cache usages are other features, which can clearly be used in detection decisions. Difference between I/O operation rate with respect to broadcast events shows a significant indicator of malicious behavior.

[4] https://developer.android.com/reference/android/content/pm/PackageManager

5.4 Advanced Threat Use-cases

It has been established that finding up-to-date malware samples is a very tedious process. Furthermore, most of the advanced attack scenarios, which are discussed in scientific researches, cannot be simulated due to the absence of practical samples. In our work, we developed different advanced malware samples that have novel attack capabilities. We demonstrated that our framework is also capable to detect future attack vectors which is likely to be seen in the practical world in the near future. This section explains multiple techniques that we have experimented during our research and how a specially crafted malware can be identified with TRAPDROID.

5.4.1 Zero-permission Malware

Applications running on the Android platform do not require any special permissions to check whether the screen is on or off. Zpware uses the weakness of this feature as an advantage to detect whenever a user is actively using the device. After detecting and active usage, malware reads all contents of the files under the /proc/net folder. These files contain metadata of all the network traffic of the device. Accessing the contents under this folder is a significant breach of the user's privacy. In addition to collecting network metadata, Zpware also processes the tools' outputs such as `getprop`, which contain more sensitive information such as device model, kernel version, etc. Whenever the user locks the screen or turns it off, the malware automatically sends all collected information to a remote server using covert channels.

It is widely known that Android applications can broadcast an Intent named by `Intent.ACTION_VIEW`[5] with a specially crafted URL argument and forces to open this URL with the default browser. In our example, we also use this technique to send collected data to a remote server in a covert fashion. There are few steps needed to successfully transfer large files since the GET requests that are called by the browser itself have a limited data header. Firstly, we need to divide the collected data into chunks. After that, we use encoding to transfer collected data chunks with the help of the `Intent.ACTION_VIEW` intent. Our sample uses the same browser page iteratively without open a new page for every intent.

[5] https://developer.android.com/reference/android/content/Intent#ACTION_VIEW

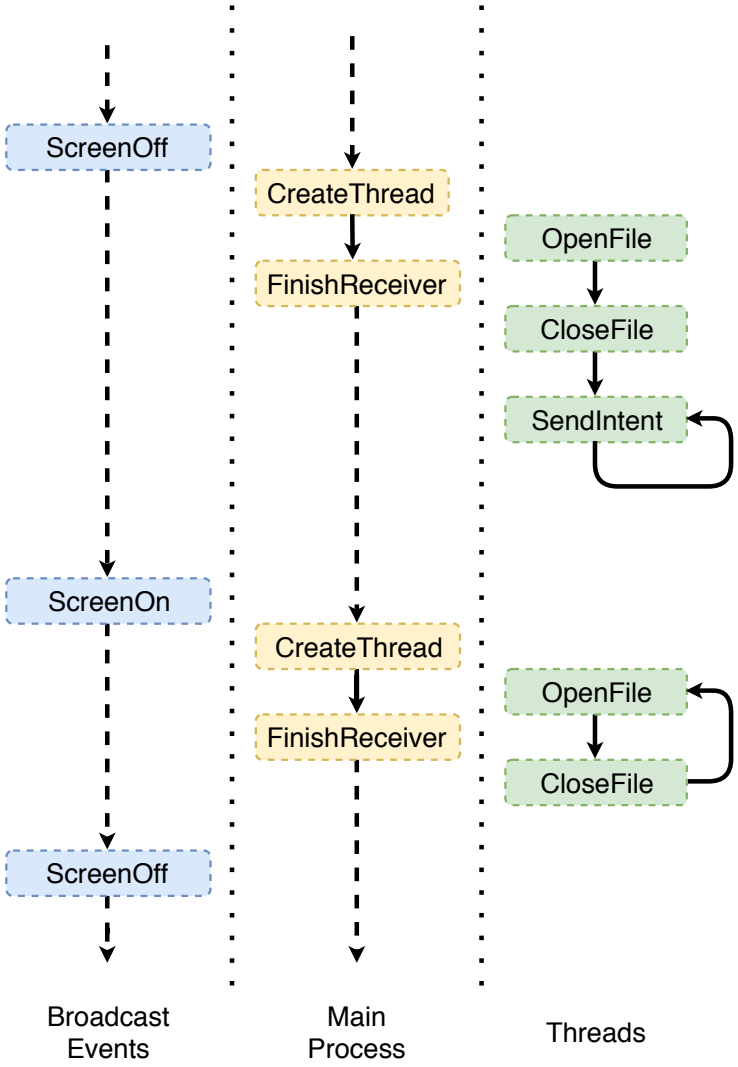

Figure 5.1 Behavior activity graph of Zpware.

We explained advanced malware, which is very hard to detect with static analysis since it does not require any permission and does not contain any suspicious API calls. It should be noted that the only feasible way to detect these types of advanced cases is through dynamic analysis. We used TRAPDROID to analyze Zpware, and Figure 5.1 shows only a small part of its behavior. It is clear to see from this figure that the application immediately starts a new thread after the screen goes off. It should also be easy to observe that the number of file system activities is relatively high when the screen goes back on. Our findings show the importance of analyzing events and their interactions with other applications in a broader picture to understand any malicious activity. Most of the malware has a trigger mechanism for their malicious actions, such as receiving an SMS from the server, checking for the existence of a file, and so forth. TRAPDROID automatically stimulates these actions and makes it very easy to analyze the behavior of these applications under different conditions.

5.4.2 Cache Attacks

Cache attacks are becoming more popular in the malware field during the past few years. Many libraries and toolkits are released to the public in order to experiment with the outcomes practically. One of the popular tools, ARMageddon(Lipp et al., 2016), is used to conduct cache attacks for the ARM platform. We could not find any practical example of malware which uses cache attacks in the Android world, so we developed one sample which uses the ARMageddon library to test our framework. Our example carries a compiled ARMageddon[6] library within itself and conducts cache-attacks to libinput.so library. It is relatively easy to capture all input activities of the user and save them into a file. After particular decision rules, the malware sends the collected data to a remote server like in the Zpware case. The primary purpose of creating such malware is not about testing the effectiveness of detecting cache attacks but to show that our framework can detect possible practical cache attacks.

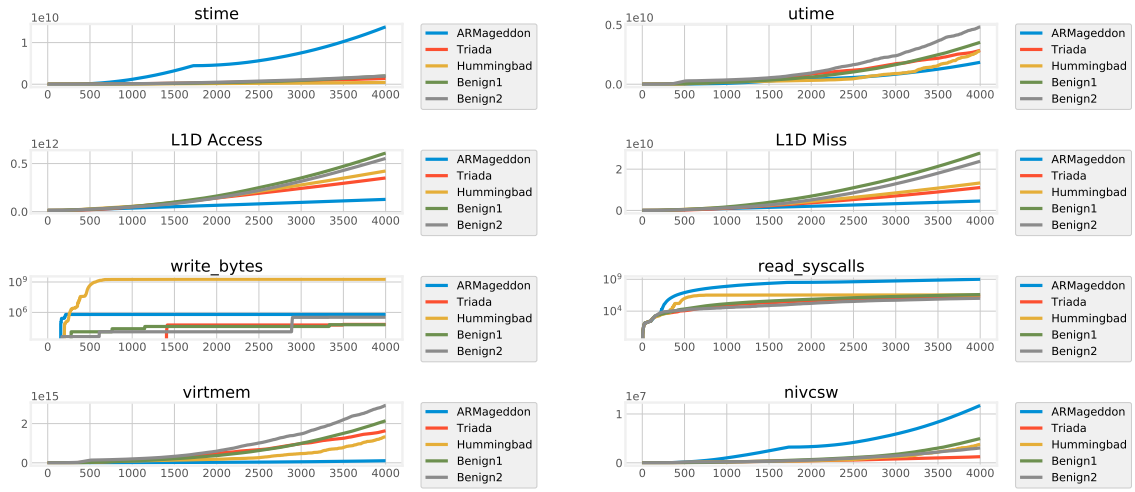

Figure 5.2 Comparison of different metrics on various samples.

In Figure 5.2, we represent some of the values of our ArmageddonApp sample during execution. One of the most critical findings is the duration of the stime in applications that executes cache-attacks. We reason that the density of the `sched_yield` system calls used during the cache-attacks increases the duration of `stime` significantly. One can argue that `sched_yield` system call is not the only way to execute a cache attack successfully. Nevertheless, our research shows that a sample cache attack scenario can also create deviations that allow our framework to detect them successfully.

[6] https://github.com/IAIK/armageddon

5.5 Remarkable Observations

Accessibility Malware Nowadays, threat actors abuse the accessibility services provided by the Android operating system itself in their malicious applications (Diao et al., 2019). Accordingly, we performed several experiments with the accessibility malware samples. The investigation has shown that these variants produce high API usage for interaction callbacks.[7] As a result, banking or similar high-profile application developers can hook the binder calls of its applications utilizing application-level interfaces to block offensive operations of accessibility services. In addition, a further study could assess the long-term effects of blocking interaction callbacks. The below code excerpt shows the simplified implementation of the disruption method.

```
public class BinderHookHandler implements InvocationHandler {
    Object base;

    public BinderHookHandler(IBinder base, Class<?> sClass) {
        try {
            Method asInterfaceMethod = sClass.getDeclaredMethod("asInterface", IBinder.class);
            this.base = asInterfaceMethod.invoke(null, base);
        } catch (Exception e) {
            throw new RuntimeException("hooked failed!");
        }
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        if ("IAccessibilityInteractionConnectionCallback".equals(method.getName())) {
            return false;
        }
        return method.invoke(base, args);
    }
}
```

Listing 5.1 Simplified implementation of the proposed method.

Dynamic IP Address Interestingly, we observed that several malware variants revealed more hidden behaviors when our framework uses the residential IP address instead of the static or VPN IP address. This result may be explained by the fact that malware developers target real people. They developed this technique to protect their infrastructure from security analysts and bypass the detection systems. As we deployed our framework to a device that contains daily usage artifacts of real people and assigned residential IP addresses, we successfully detected this evasive method. A reasonable approach to tackle this issue could be using real people's devices as a detection environment.

[7] Full name of the call: "android.view.accessibility.IAccessibilityInteractionConnectionCallback"

6. CONCLUSION

Analyzing malware in a bare-metal environment offers a massive advantage over emulation. Malware continuously becomes more capable of detecting and changing execution flows in emulated environments. We present TRAPDROID as a practical example of analyzing different Android malware types by utilizing unified behavior profiles instead of using a single system call metric. Besides the malware analysis framework, we implemented several advanced malware and analyzed their behavioral characteristics in our framework. Although our detection system shows many exciting results, more research is needed to understand kernel-level properties such as cache usage, timing differences, and battery consumption.

We employed state-of-the-art machine learning algorithms and novel deep-learning models to identify malicious applications upon developing the analysis framework. Then, we evaluated the proposed models' accuracy and performance on the up-to-date dataset, which contains widely known variants, custom-crafted applications, and 0-day and 1-day samples collected from different research datasets and private resources. With UBF-P, UBF-R, and UBF-A tokenization schemes, the GB (Gradient Boosted Trees) classification algorithm yields 97.75%, 97.92%, and 98.08% F-1 scores, respectively. Moreover, we reached 98.33% accuracy with a deep-neural network developed using CNN with a single embedding layer. Later, we populated annotations for each sequence to another different embedding layer, leading us to reach 98.42% F-1 and 98.41% accuracy scores on a balanced (MWR=0.5) dataset containing 6000 applications. The empirical findings in this study provide a new understanding of treating dynamic malware analysis as a text-classification problem.

The main weakness of this study was the resource and time limitations of the bare-metal environments. Further investigation and experimentation into speeding up the restoration and stimulation is strongly recommended. In general, therefore, it seems that any improvements on the stimulation will also improve the performance of the detection.

BIBLIOGRAPHY

Abderrahmane, A., Adnane, G., Yacine, C., & Khireddine, G. (2019). Android malware detection based on system calls analysis and cnn classification. In *2019 IEEE Wireless Communications and Networking Conference Workshop (WCNCW)*, (pp. 1–6). IEEE.

Alptekin, H., Yildizli, C., Savas, E., & Levi, A. (2019). Trapdroid: Bare-metal android malware behavior analysis framework. In *2019 21st International Conference on Advanced Communication Technology (ICACT)*, (pp. 664–671). IEEE.

Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H., Rieck, K., & Siemens, C. (2014). Drebin: Effective and explainable detection of android malware in your pocket. In *Ndss*, volume 14, (pp. 23–26).

Aslan, Ö. A. & Samet, R. (2020). A comprehensive review on malware detection approaches. *IEEE Access*, *8*, 6249–6271.

Au, K. W. Y., Zhou, Y. F., Huang, Z., & Lie, D. (2012). Pscout: analyzing the android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security*, (pp. 217–228).

Bello, L. & Pistoia, M. (2018). Ares: triggering payload of evasive android malware. In *2018 IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, (pp. 2–12). IEEE.

Bulazel, A. & Yener, B. (2017). A survey on automated dynamic malware analysis evasion and counter-evasion: Pc, mobile, and web. In *Proceedings of the 1st Reversing and Offensive-oriented Trends Symposium*, (pp. 1–21).

Burguera, I., Zurutuza, U., & Nadjm-Tehrani, S. (2011). Crowdroid: behavior-based malware detection system for android. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, (pp. 15–26).

Canfora, G., Medvet, E., Mercaldo, F., & Visaggio, C. A. (2015). Detecting android malware using sequences of system calls. In *Proceedings of the 3rd International Workshop on Software Development Lifecycle for Mobile*, (pp. 13–20).

Chaurasia, P. (2015). *Dynamic analysis of Android malware using DroidBox*. PhD thesis, Tennessee State University.

Diao, W., Zhang, Y., Zhang, L., Li, Z., Xu, F., Pan, X., Liu, X., Weng, J., Zhang, K., & Wang, X. (2019). Kindness is a risky business: on the usage of the accessibility apis in android. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, (pp. 261–275).

Feng, Y., Anand, S., Dillig, I., & Aiken, A. (2014). Apposcopy: Semantics-based detection of android malware through static analysis. In *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*, (pp. 576–587).

Hou, S., Saas, A., Chen, L., & Ye, Y. (2016). Deep4maldroid: A deep learning framework for android malware detection based on linux kernel system call graphs. In *2016 IEEE/WIC/ACM International Conference on Web Intelligence Workshops (WIW)*, (pp. 104–111). IEEE.

Jing, Y., Zhao, Z., Ahn, G.-J., & Hu, H. (2014). Morpheus: automatically generating heuristics to detect android emulators. In *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC)*, (pp. 216–225).

Kabakus, A. T. & Dogru, I. A. (2018). An in-depth analysis of android malware using hybrid techniques. *Digital Investigation*, *24*, 25–33.

Karbab, E. B., Debbabi, M., Alrabaee, S., & Mouheb, D. (2016). Dysign: dynamic fingerprinting for the automatic detection of android malware. In *2016 11th International Conference on Malicious and Unwanted Software (MALWARE)*, (pp. 1–8). IEEE.

Kouliaridis, V., Barmpatsalou, K., Kambourakis, G., & Chen, S. (2020). A survey on mobile malware detection techniques. *IEICE Transactions on Information and Systems*, *103*(2), 204–211.

Li, J., Sun, L., Yan, Q., Li, Z., Srisa-An, W., & Ye, H. (2018). Significant permission identification for machine-learning-based android malware detection. *IEEE Transactions on Industrial Informatics*, *14*(7), 3216–3225.

Li, Y., Yang, Z., Guo, Y., & Chen, X. (2017). Droidbot: a lightweight ui-guided test input generator for android. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, (pp. 23–26). IEEE.

Lipp, M., Gruss, D., Spreitzer, R., Maurice, C., & Mangard, S. (2016). Armageddon: Cache attacks on mobile devices. In *25th USENIX Security Symposium (USENIX Security 16)*, (pp. 549–564).

Mahdavifar, S., Kadir, A. F. A., Fatemi, R., Alhadidi, D., & Ghorbani, A. A. (2020). Dynamic android malware category classification using semi-supervised deep learning. In *2020 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech)*, (pp. 515–522). IEEE.

Mutti, S., Fratantonio, Y., Bianchi, A., Invernizzi, L., Corbetta, J., Kirat, D., Kruegel, C., & Vigna, G. (2015). Baredroid: Large-scale analysis of android apps on real devices. In *Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC)*, (pp. 71–80).

Olson, R. S., Bartley, N., Urbanowicz, R. J., & Moore, J. H. (2016). Evaluation of a tree-based pipeline optimization tool for automating data science. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, GECCO '16, (pp. 485–492)., New York, NY, USA. ACM.

Park, J., Chau, N.-T., Nguyen-Vu, L., Yoon, J., & Jung, S. (2020). A-pot: A comprehensive android analysis platform based on container technology. *IEEE Access*, *8*, 199638–199645.

Prusa, J. D. & Khoshgoftaar, T. M. (2016). Comparing approaches for combining data sampling and feature selection to address key data quality issues in tweet sentiment analysis. In *The Twenty-Ninth International Flairs Conference*.

Qiu, J., Zhang, J., Luo, W., Pan, L., Nepal, S., & Xiang, Y. (2020). A survey of android malware detection with deep neural models. *ACM Computing Surveys (CSUR)*, *53*(6), 1–36.

Surendran, R., Thomas, T., & Emmanuel, S. (2020). On existence of common malicious system call codes in android malware families. *IEEE Transactions on Reliability*, *70*(1), 248–260.

Tam, K., Fattori, A., Khan, S., & Cavallaro, L. (2015). Copperdroid: Automatic reconstruction of android malware behaviors. In *NDSS Symposium 2015*, (pp. 1–15).

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). Attention is all you need. In *Advances in neural information processing systems*, (pp. 5998–6008).

Vinayakumar, R., Soman, K., Poornachandran, P., & Sachin Kumar, S. (2018). Detecting android malware using long short-term memory (lstm). *Journal of Intelligent & Fuzzy Systems*, *34*(3), 1277–1288.

Wang, Q., Li, B., Xiao, T., Zhu, J., Li, C., Wong, D. F., & Chao, L. S. (2019). Learning deep transformer models for machine translation. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, (pp. 1810–1822).

Wang, Z., Liu, Q., & Chi, Y. (2020). Review of android malware detection based on deep learning. *IEEE Access*, *8*, 181102–181126.

Weichselbaum, L., Neugschwandtner, M., Lindorfer, M., Fratantonio, Y., Van Der Veen, V., & Platzer, C. (2014). Andrubis: Android malware under the magnifying glass.

Xiao, X., Zhang, S., Mercaldo, F., Hu, G., & Sangaiah, A. K. (2019). Android malware detection based on system call sequences and lstm. *Multimedia Tools and Applications*, *78*(4), 3979–3999.

Yan, L. K. & Yin, H. (2012). Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *21st USENIX Security Symposium (USENIX Security 12)*, (pp. 569–584).

Yeh, C.-W., Yeh, W.-T., Hung, S.-H., & Lin, C.-T. (2016). Flattened data in convolutional neural networks: Using malware detection as case study. In *Proceedings of the International Conference on Research in Adaptive and Convergent Systems*, (pp. 130–135).

Zheng, M., Sun, M., & Lui, J. C. (2014). Droidtrace: A ptrace based android dynamic analysis system with forward execution capability. In *2014 international wireless communications and mobile computing conference (IWCMC)*, (pp. 128–133). IEEE.