# Making the Case for Stealthy, Reliable, and Low-overhead Android Malware Detection and Classification

*A thesis submitted*

*in partial fulfilment of the requirements*

*for the degree of*

Doctor of Philosophy

*by*

**Saurabh Kumar**

**15211267**

*to the*


**DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING**

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

June 2022

# Certificate

It is certified that the work contained in this thesis entitled **"Making the Case for Stealthy, Reliable, and Low-overhead Android Malware Detection and Classification"** by **"Saurabh Kumar"** has been carried out under our supervision and that this work has not been submitted elsewhere for a degree.

Prof. Sandeep K. Shukla
Professor
Department of Computer Science & Engineering
Indian Institute of Technology Kanpur

Prof. Biswabandan Panda
Assistant Professor
Department of Computer Science & Engineering
Indian Institute of Technology Bombay

*June 2022*

# Declaration

This is certified that the thesis entitled **"Making the Case for Stealthy, Reliable, and Low-overhead Android Malware Detection and Classification"** has been authored by me. It presents the research conducted by me under the supervision of **Prof. Sandeep K. Shukla (IITK)** and **Prof. Biswabandan Panda (IITB)**. To the best of my knowledge, it is an original work, both in terms of research content and narrative, and has not been submitted elsewhere, in part or in full, for a degree. Further, due credit has been attributed to the relevant state-of-the-art and collaboration (if any) with appropriate citations and acknowledgements, in line with established norms and practices.

Saurabh Kumar
Program: Doctor of Philosophy
Department of Computer Science & Engineering
Indian Institute of Technology Kanpur
Kanpur 208016

*June 2022*

# *Abstract*

---

Name of the student: **Saurabh Kumar** Roll No: **15211267**

Degree for which submitted: **Doctor of Philosophy**

Department: **Computer Science & Engineering**

Thesis title: **Making the Case for Stealthy, Reliable, and Low-overhead Android Malware Detection and Classification**

Name of Thesis supervisors:

1. **Prof. Sandeep K. Shukla**, Professor, Dept. of CSE, IIT Kanpur
2. **Prof. Biswabandan Panda**, Assistant Professor, Dept. of CSE, IIT Bombay

Month and year of thesis submission: **June 2022**

---

The increased popularity and wide adoption of Android as a mobile OS platform have rendered the platform a primary target for malware attackers. Because of the rapid increase in malware numbers, variations, and diversity, detection and classification of Android malware have become challenging. In the recent years, a large number of automatic malware detection and classification systems have been evolved to deal with the dynamic nature of malware growth. Much of these are based on employing static, dynamic, or both analysis methodologies. Even though a good number of malware analysis systems are available, malware often finds its way to be unleashed into a device, bypassing the defense system of the application store (Play Store). It is conceivable because the Play Store's security systems based on existing dynamic analysis frameworks have two fundamental flaws: they lack flawless anti-emulation-detection protection and efficient cross-layer profiling capabilities. Additionally, Android allows the installation of an application from unverified sources auch as third-party markets or sideloading, enabling another way to infect a mobile device.

Moreover, malware detection has received more attention than family identification in the past, which is also essential to speed up recognizing and mitigating a known strain. This thesis develops strategies for detecting and classifying Android malware that is stealthy and reliable. These strategies when implemented are low-overhead.

We first present InviSeal, a comprehensive and scalable dynamic analysis framework with low-overhead cross-layer profiling approaches and a detailed anti-emulation-detection mechanism to protect the defense system of an application store. We empirically demonstrate that InviSeal has a very low-overhead when compared to the existing approaches to achieving cross-layer profiling. We also show the capability of InviSeal to thwart emulation-detection by malware and detect collusion attacks through cross-layer profiling.

Next, we present an on-device malware detection system (DeepDetect). DeepDetect employs a machine learning model based on static features since a foolproof security system of an application store does not stop users from installing applications from untrusted sources. DeepDetect takes ∼5.32 seconds to identify an Android application as malware on a real device while achieving a more than 97% malware detection rate. Furthermore, DeepDetect consumes 0.45% of total device energy in analyzing 50 applications.

Finally, we present MAPFam, a malware family classification framework that learns a machine learning model to categorize Android malware into families. The proposed family identification framework achieves more than 97% accuracy for the top 60 malware families with a 97.55% model reliability rate. Moreover, the MAPFam model can perfectly identify 36 malware families out of 60.

# *Acknowledgements*

I would like to express my sincere gratitude to my thesis advisors Prof. Sandeep K. Shukla and Prof. Biswabandan Panda for their constant support and guidance. Their inputs throughout the course of the thesis work were extremely beneficial and constructive.

I would also like to thank Prof. Debadatta Mishra for his valuable comments and feedback, which was extremely useful for improving my research work.

I would like to acknowledge my fellow graduate students and members of KD-222 lab for their feedback.

I would also like to thank my family. Without their support and encouragement, this accomplishment would not have been possible. I would also like to thank my friends for always being there whenever I needed them.

# Contents

# List of Figures

# List of Tables

# Abbreviations

| | |
|---|---|
| **Acc** | **Acc**uracy |
| **ADB** | **A**ndroid **D**ebug **B**ridge |
| **AOSP** | **A**ndroid **O**pen **S**ource **P**roject |
| **API** | **A**pplication **P**rogramming **I**nterface |
| **APK** | **A**ndroid **P**ac**k**age |
| **App** | **App**lication |
| **ART** | **A**ndroid **R**untime |
| **AT** | **AT**tention |
| **AUC** | **A**rea **U**nder the **C**urve |
| **AV** | **A**nti**V**irus |
| **AVD** | **A**ndroid **V**irtual **D**evice |
| **BTS** | **B**ase **T**ransceiver **S**tation |
| **DIFT** | **D**ynamic **I**nformation **F**low **T**racking |
| **CID** | **C**ell **ID** |
| **CM** | **C**affeine**M**ark |
| **DT** | **D**ecision **T**ree |
| **DVM** | **D**alvik **V**irtual **M**achine |
| **ELF** | **E**xecutable and **L**inkable **F**ormat |
| **ET** | **E**xtra **T**ree |
| **F1** | **F1** Score |
| **FPR** | **F**alse **P**ositive **R**ate |

| | |
|---|---|
| **GPS** | **G**lobal **P**ositioning **S**ystem |
| **HAL** | **H**ardware **A**bstraction **L**ayer |
| **ID** | **Id**entification |
| **IDC** | **I**nternational **D**ata **C**orporation |
| **IMEI** | **I**nternational **M**obile **E**quipment **I**dentity |
| **IMSI** | **I**nternational **M**obile **S**ubscriber **I**dentity |
| **JNI** | **J**ava **Native Interface** |
| **JVM** | **J**ava **V**irtual **M**achine |
| **LAC** | **L**ocation **A**rea **C**ode |
| **LiME** | **Li**nux **M**emory **E**xtractor |
| **LR** | **L**ogistic **R**egression |
| **MCC** | **M**obile **C**ountry **C**ode |
| **MNC** | **M**obile **N**etwork **C**ode |
| **NDK** | **N**ative **D**evelopment **K**it |
| **NN** | **N**eural **N**etwork |
| **NOP** | **N**o **Op**eration |
| **Opc** | **Opc**odes |
| **OS** | **O**perating **S**ystem |
| **PID** | **P**rocess **ID** |
| **PM** | **P**ackage **M**anager |
| **Pre** | **Pre**cision |
| **RA** | **R**estricted **A**PIs |
| **Rec** | **Rec**all |
| **RF** | **R**andom **F**orest |
| **RFE** | **R**ecursive **F**eature **E**limination |
| **RFECV** | **R**ecursive **F**eature **E**limination with **C**ross **V**alidation |
| **ROC** | **R**eceiver **O**perating **C**haracteristic |
| **RP** | **R**equested **P**ermission |
| **SA** | **S**ensitive **A**PIs |

| | |
|---|---|
| **SC** | **S**ystem **C**ommands |
| **SDK** | **S**oftware **D**evelopment **K**it |
| **SIM** | **S**ubscriber **I**dentification **M**odule |
| **STDNeut** | **S**ensor, **T**elephony system, and **D**evice state information **Neut**ralizer |
| **TAC** | **T**ype **A**llocation **C**ode |
| **TPR** | **T**rue **P**ositive **R**ate |
| **UDI** | **U**nique **D**evice **I**nformation |
| **UID** | **U**ser **ID** |
| **UP** | **U**sed **P**ermission |
| **VH** | **V**oting classifier in **H**ard mode |
| **VM** | **V**irtual **M**achine |
| **VS** | **V**oting classifier in **S**oft mode |
| **WB** | **W**eb **B**rowser |

# Chapter 1

# Introduction

In recent years, Android has become one of the most popular operating systems (OSes) for smartphones because of its open-source nature and large support for different applications (apps). Recently, a report shared by the International Data Corporation (IDC) USA for smartphone OSes shows that the Androids' market share in the second quarter of 2021 was 83.8%, and predicts that it will acquire 84.9% of the market share by 2025 [4]. Because it is open-source and easy to use, it has piqued the interest of manufacturers worldwide in producing inexpensive mobile devices compared to other platforms. Furthermore, Android is gaining popularity in devices other than smartphones, including Tablets, TVs, Wearables, and, most recently, IoT devices. Moreover, the simplicity of the Android framework with regard to app development has resulted in substantial growth in the number of mobile apps developed worldwide. A study from Statista shows that every day more than 3.5K Android apps were released in the year 2020 [5].

## 1.1 Motivation

With the large-scale adaptation of Android OS and ever-increasing contributions in the Android app space, security has become a non-trivial challenge recently. According to the AV-TEST study, approx 3.39 million new Android malware (see Figure 1.1) were

discovered in 2021 [1]. This shows that more than 9.2K new malware for the Android platform was created each day in 2021 [1]. Because of the rapid development in malware numbers, variations, and diversity, detection and classification of Android malware have become challenging.

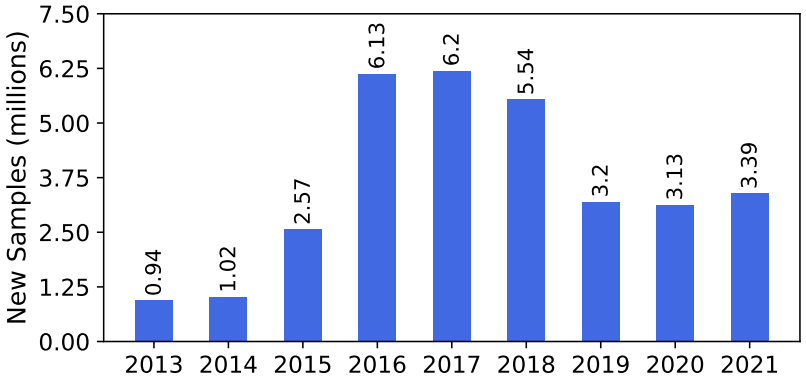



FIGURE 1.1: Development of new Android malware every year [1].

## 1.2 Problem

In recent years, a large number of automatic malware detection and classification systems ([6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18]) have evolved to tackle the dynamic nature of malware growth using either static, dynamic, or both analysis techniques [19]. These systems generally operate offline on a server machine and are deployed on an app store like Google Play Store. Even though a vast majority of malware analysis systems are available, malware often finds a way to be unleashed into a device bypassing the defense system of an app store. For example, Google Play Protect which is used to certify Android apps, fails to detect malware that spread across 85 different apps affecting nine million Android devices [20]. The possible reasons using which malware may infect a device are as follows:

***(i)*** Defense systems built around existing dynamic analysis frameworks deployed on the app store suffer from two major issues. First, they do not provide foolproof anti-emulation-detection measures. Second, they lack efficient cross-layer profiling capabilities. Malware generally exploits these vulnerabilities to bypass the defense

tactics adopted by the app stores to enter the market and subsequently into a device. Therefore, a stealthy dynamic analysis system is needed so that malware writers find it difficult to bypass the defense system of an app store.

***(ii)*** Android allows the installation of an app from unverified sources (e.g., third-party market and sideloading), which opens up other ways for malware to infect the smartphones. Therefore, deploying the security system only at the app store does not solve the entire problem. Hence, there is a need for a low-overhead on-device malware detection system.

***(iii)*** Moreover, identifying the malware family is as important as detecting the malware. Malware detection has traditionally received more attention than family identification. Aside from detecting malware, classifying the malware's family allows security analysts to reuse malware removal techniques that have been proven to work for that family of malware. Family information also helps in the articulation of the damages done by malware. Therefore, we must pay equal attention to malware family identification as we do to malware detection.

## 1.3 Our Goal

We believe that if a stealthy, reliable, and low-overhead malware detection and classification process are present. It will automatically address the present scale and dynamic nature of malware growth. Therefore, we should concentrate on developing a process/system that makes it difficult for a clever malware creator to infiltrate the app store and mobile devices. Moreover, we would like to emphasize the need for a validation mechanism to understand the effectiveness of the same.

## 1.4 Our approach

To make the case of stealthy, reliable, and low-overhead malware detection and classification process, we use the following approach:

*(i)* **Validation Mechanism:** We design an emulation-detection library that can arm several existing and new malware to study the efficacy of existing dynamic analysis frameworks. Furthermore, with the outdated dataset, it is hard to measure the efficiency of malware detectors with the current state of app design strategy and evasion techniques. Hence, we create multiple datasets that reflect the current state of evasion techniques and the app design paradigm.

*(ii)* **Stealthy dynamic analysis system:** As discussed earlier, malware detection is generally performed by the app stores in an offline manner using the emulated platform. We use emulation-detection library to arm several existing malware with different levels of emulation-detection capabilities and study the efficacy of anti-emulation-detection measures of well known dynamic analysis frameworks. Further, using the findings of our analysis, we develop a stealthy dynamic analysis framework with cross-layer profiling capabilities.

*(iii)* **Low-overhead on-device malware detection:** We carefully select relevant features using a feature engineering framework to create a lightweight malware detection model to design a low-overhead on-device malware detector. Additionally, most of the time-consuming steps in malware detection are feature extraction processes. Hence, we design an energy-efficient feature extraction module for a real device. Finally, we build an on-device malware detector and evaluate its efficacy against multiple datasets, including obfuscated and Pegasus malware samples.

*(iv)* **Reliable family classification framework:** To create a reliable malware family classification framework, we experimentally evaluate the efficacy and reliability of

different features. We then design an accurate and reliable malware family classification framework by considering the best features, followed by an evaluation of the final framework.

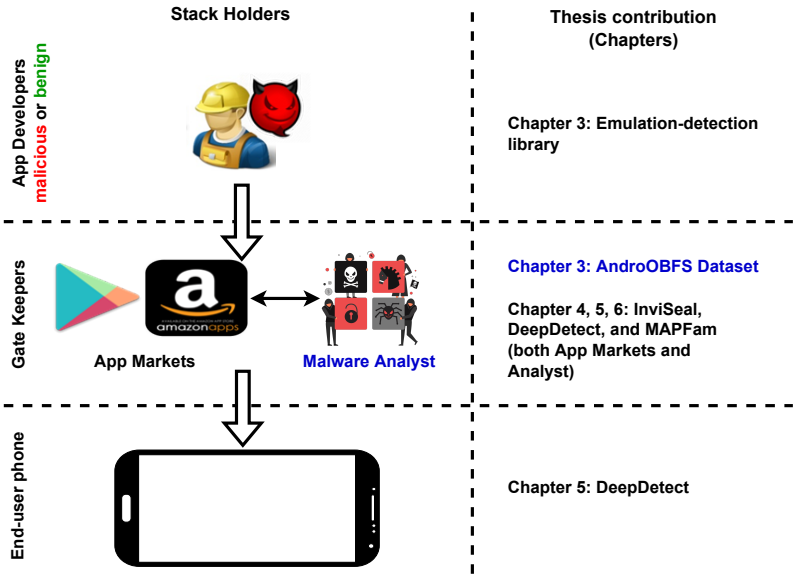



FIGURE 1.2: Stack holders of the system that are benefiting from the different contributions of this thesis (Chapter-wise).

## 1.5 Contributions

In this thesis, we make four major contributions. Figure 1.2 shows the different stack holders of the systems that benefit from the contributions of this thesis. The contribution that we made are as follows:

***(i)*** We design an emulation-detection library encompassing several advanced detection techniques like distributed detection and GPS information. The library can be configured with varying levels of emulation-detection methods and can be seamlessly embedded into different malware. Furthermore, we create multiple datasets, including an up-to-date obfuscated malware and all the potential biases-free dataset. We also create a dataset that includes samples of Pegasus malware. (Chapter 3)

*(ii)* **InviSeal, a stealthy dynamic analysis framework:** We design InviSeal, a comprehensive and scalable dynamic analysis framework that includes low-overhead cross-layer profiling techniques and detailed anti-emulation-detection measures along with the basic emulation features. While providing an emulator-based comprehensive analysis platform, InviSeal strives to remain behind-the-scenes to avoid emulation-detection. We empirically demonstrate that the proposed OS layer profiling utility to achieve cross-layer profiling incurs very little performance overhead compared to existing approaches. Further, we show the capability of InviSeal to thwart emulation detection by malware and detect collusion attacks through cross-layer profiling by using several use cases and experiments. (Chapter 4)

*(iii)* **DeepDetect, low-overhead on-device malware detector:** DeepDetect enables on-device malware detection by employing a machine learning based model on static features. With effective feature engineering, DeepDetect can be used on-device. To classify an Android app as malware, it takes ∼5.32 seconds, which is 2.23X faster than API based malware detector, while consuming 0.45% (for 50 apps) of total device energy. DeepDetect provides a malware detection rate of 99.9% for known malware with a 0.01% false-positive rate. For unseen/new samples, it detects more than 97% malware with a false-positive rate of 1.73%. Further, in the presence of obfuscated malware, DeepDetect correctly detects 95.57% of malware samples. We also evaluate our model against the Pegasus malware sample and with a new dataset after removing the potential biases across space and time. (Chapter 5)

*(iv)* **MAPFam, malware family classification framework:** The main aim of this work is to classify an Android malware into its family. This work is premised on a starting hypothesis that features extracted from API packages rather than API calls lead to more precise classification. Our experiments indeed shows that API package based model provides ∼1.63X more accurate classification compared to an API call based method. Our machine learning based malware family classification system uses API packages, requested permissions, and other features from the Manifest files. The proposed family classification system achieves accuracy and average precision above

97% for the top 60 malware families by using only 81 features with 97.55% of model reliability rate (Kappa score). The experimental results also shows that MAPFam can perfectly identity 36 malware families. (Chapter 6)

## 1.6 Thesis Organization

This thesis is organized into seven chapters, including this one, which introduces the topic and states the problem. The rest of the thesis is structured as follows.

Chapter 2 discusses the background of Android OS, Android application package, and security architecture. After that, it introduces mobile malware, malware analysis techniques, and limitations. Finally, it made some observations that serve as the foundation key points for this thesis.

Chapter 3 elaborates on the design of the emulation-detection library and datasets created to evaluate an analysis framework.

Chapter 4 shows the empirical analysis of the existing dynamic analysis framework against the emulation-detection library. Then it presents InviSeal, a stealthy dynamic analysis framework with cross-layer profiling capabilities.

Chapter 5 discusses the design of DeepDetect, an on-device malware detector. First, it discusses the challenges of designing such a system and how to handle them. Finally, it presents DeepDetect, followed by its evaluation.

Chapter 6 presents a malware family classification framework MAPFam, and why such a system is required.

Finally, Chapter 7 concludes this thesis and provides directions for future work.

# Chapter 2

# Background and Related Work

Android is an open-source mobile OS based on the Linux kernel, which Google has developed. This chapter provides an overview of the Android OS & app, mobile malware & their evolution, tools & techniques to analyze them, and counter analysis techniques. At last, we provide some observations based on our study on malware analysis techniques.

## 2.1 Android Background

In this section, we discuss the Android platform architecture, security architecture of Android, and Android application package. After that, we provide an overview of the Base Transceiver Station (BTS) as a smartphone frequently communicates with it.

### 2.1.1 Android Platform Architecture

Android is a Linux-based open-source software stack for mobile devices. The Android platform comprises of six major components (Figure 2.1): the Linux Kernel, Hardware Abstraction Layer (HAL), native libraries, Android runtime, Application framework (Java API Framework), and Application layer (default apps or third-party apps) [2]. Working of each component is as follows:

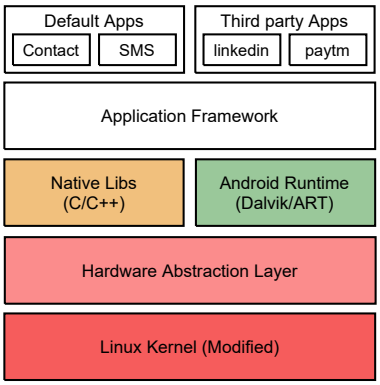



FIGURE 2.1: Android Platform Layered Architecture [2].

***(i)* Linux Kernel:** Linux OS (a.k.a. Linux Kernel) is the basis of the Android Platform. The Android virtual machine (ART/DVM) depends on the Linux Kernel for essential functions like managing memory, threads, power management, etc. Android uses many of Linux's key security features, such as process-level isolation, user-group-based file permissions for separating privileges, etc.

***(ii)* Hardware Abstraction Layer (HAL):** The HAL exposes device hardware capabilities to the Application Framework through standard interfaces. The Linux kernel system call APIs are used to implement the HAL layer. When a framework API makes its first call to access the device hardware, the Android system loads the corresponding library module.

***(iii)* Native Libraries:** Many essential Android features and services, such as ART and access to HAL APIs, rely on methods supplied by native C/C++ libraries. The Android platform also has Java framework APIs that allow Apps to access the functionality of some of these native libraries. Some of these Java framework APIs include media server framework, SQLite Database, and Libs (bionic).

***(iv)* Android Runtime:** It is the app runtime of the Android, a virtual machine similar to JVM. The difference between Android Virtual Machine (ART) and JVM is as follows: JVM is a stack based virtual machine whereas ART is a register based. Therefore, ART incurs less performance overhead as compared to the JVM. Android uses two types of

virtual machines: Dalvik virtual machine (Android version $< 5$) and ART (recent Android releases).

***(v)* Application Framework:** It provides the Java API to facilitate app development in the Android mobile platform. For example, application framework provides APIs to use view system objects like button, text view, list control etc. It also provides APIs to access information from resource manager, activity manager etc.

***(vi)* Application Layer:** This layer contains actual Android apps. There are two types of apps available with Android: system (default) apps and third-party apps. System apps are developed by device vendor and shipped with the device (e.g., contacts, browser). On the other hand, third-party apps are the custom build apps and hosted on the market (play store) from where a device user can install depending on her requirements (e.g., WhatsApp and Uber). Note that, system Apps have higher privileges compared to the third-party apps.

### 2.1.2 Android Application Package (APK)

The Android application package is commonly known as an APK. An app developer uses Android Studio or another toolchain (Cordova [21], Ionic framework [22], and others) to create Android apps. The final result of the app development process is an APK. It comprises of five major components [23], as illustrated in Figure 2.2. The description of these components are follows.


FIGURE 2.2: Android application package (APK) and its components

***(i)* Dex File (Classes.dex):** Dex file stores the main execution logic of an app. The entire logic of an app can be kept in a single Dex file or split across multiple Dex files. Generally, a Dex file is made up of four components, which are as follows:

- **Activities:** An activity represents a single screen through which a user can interact with the app.
- **Content Providers:** A content provider supplies data from one app to another on request.
- **Services:** These are the processes that keep running in the background without any need for interaction with the user. These services work even when the app is not running.
- **Broadcast Receivers:** A broadcast receiver listens for messages sent out by other apps or the system itself. People often call these messages events or intents. For example, if our phone receives a short message service (SMS), a broadcast receiver can listen to it and do something in response to the SMS.

***(ii)* Native Libs:** At times, a piece of code written in C/C++ is included in the native library. To invoke a native function in an Android app, Java Native Interface (JNI) is used.

***(iii)* Resources:** Elements such as images, strings, color value, etc., used in an app fall under the category of resources.

***(iv)* META_INF:** It contains meta-information about an app. It also holds the public key of the signing certificate, which is used to verify the integrity of an app.

***(v)* Application Manifest:** Each app contains an Application manifest file named as *AndroidManifest.xml* [24]. It stores all the information about an app like receivers, content providers, activities, permissions, capabilities used, etc. All components need to be registered in this file. If a component is not present in the Manifest file, then its functionality would not be visible to Android.

### 2.1.3 Android Security Architecture

In this section, we look into the security features provided by the Android OS regarding Application Isolation (Sandbox) and the concept of Permissions and least privilege.

#### 2.1.3.1 Application Sandbox

The Application Sandbox governs an app's access right to the system resources. Each app in Android executes in an isolated environment, so it cannot access the data of other apps. Proper access rights are required if an app wants to access other app resources. Generally, access rights to other resources are associated with the permission. Hence, individual permissions must be declared in the Android manifest file.

Android uses the Linux user ID model for the app isolation. Each app runs as a separate user assigned to it during the installation. Android forks a new virtual machine (ART/DVM) when an app is ready for execution, and the corresponding user ID is then associated with the forked virtual machine. Figure 2.3 shows the Android sandbox architecture.

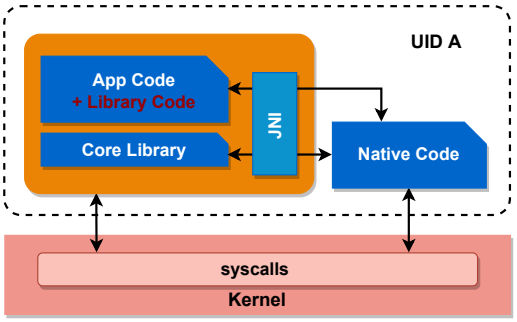



FIGURE 2.3: Application sandboxing in Android

In Figure. 2.3, the Classes.dex file contains executable dex code that ART executes. This code is generally written in Java/Kotlin and converted into the dex bytecode so that ART can execute it. If an app contains native code, this code lies outside the virtual machine boundary. The ART takes the help of Java Native Interface (JNI) to use the functionality available in native code. JNI is the bridge between Java/Kotlin and native code.

#### 2.1.3.2 Permissions

To access other resources which do not belong to an app is provided on the basis of permissions. In Android, each permission has a protection level, which describes a protocol for granting permission to the app. Permissions protection level is classified into four categories, as follows [25]:

- **Normal:** These permissions are given to an app when an app gets installed. These permissions protect such resources that are not related to user privacy.
- **Dangerous:** These permissions pertain to resources that directly or indirectly infringe on the privacy of end-users, such as GPS and SMS. For Android versions less than 6.0, these permissions are granted to an app at the time of installation, but from version 6.0 and above, a user can adjust these permissions after app installation.
- **Signature:** Android grants these permissions only if the app asking for them is signed with the same certificate as the app that declared the permission.
- **SignatureOrSystem:** These permissions are similar to the signature protection level, except that the app signing certificate is the same as the certificate used to sign the Android OS image.

### 2.1.4 Base Transceiver Station (BTS)

BTS [26] is a piece of wireless communication equipment that establishes communication between a mobile device and a network. The BTS is associated with a base station ID that uniquely identifies a BTS worldwide. Base station ID comprises of four components: *(i)* mobile country code (MCC), *(ii)* mobile network code (MNC), *(iii)* location area code (LAC), and *(iv)* a cell ID (CID). A combination of these gives a unique identity to a BTS. Several commercial and public services are available which provides the geo-location of a cell by submitting its station's unique ID [27].

## 2.2 Android Malware

With the large-scale adaptation of Android OS and ever-increasing contributions in the Android app space, the focus of attackers and malware authors on the Android platform has also increased significantly. Android malware primarily targets smartphones to infiltrate users' sensitive data or to cause inconvenience by remotely taking control of the infected device and subscribing to premium services. It can also involve infected devices in large-scale attacks such as Denial of Service (DoS). Another concern is ransomware, which encrypts users' data and demands a ransom payment to restore access. In Android, the permissions system defines access rights to the resources (as previously discussed). Most Android malware writers utilize phishing or social engineering to deceive users into accepting permissions that are then misused by the malicious app. Furthermore, malware writers are taking the task of creating malware to new heights. Nowadays, malware are becoming more sophisticated and impactful. These malware are disguised as harmless and useful apps. Following are some common types of malware [28]:

- Worm - Worms are a type of malware that replicates itself in order to infect other devices. Worms do not require any user interaction to execute.
- Ransomware - Ransomware encrypts data, rendering it inaccessible and unusable. A ransom is then demanded in order to gain access.
- Spyware - Spyware can secretly monitor a user's activity, steal valuable information, and send it to another entity that may harm the user.
- Adware - These programs pop up unpleasant advertising banners on the user interface to generate money based on clicks and downloads. Nowadays, malware writers create malvertising codes that steal valuable information from infected devices and root them, allowing them to do tasks like downloading particular adware.
- Trojan - A Trojan infiltrates a user's computer by masquerading as a legitimate program that the user willingly downloaded.

- Virus - A virus is a type of malware which when executed infects other files by inserting malicious code.
- Expander - Expander harms the user by increasing the billing amounts.
- Backdoor - Backdoor is a piece of malicious code that allows unauthorized access to the infected devices.

One of the few ways to keep such malicious software out of smartphones is to install apps only from trusted sources and avoid visiting suspicious emails and links. Another important measure is to keep the Android OS up to date with all updates and security patches. Each Android release brings a slew of new security features. In response to the alarming increase in fraudulent apps, Google has launched Bouncer [29], a security service that automatically scans apps in the background and is responsible for a 40% reduction in harmful apps on the app store. Despite various attempts, they still make their way to the mass market and frequently go unnoticed by traditional signature-based anti-virus software. Next, we discuss the malware evolution in the mobile platform.

### 2.2.1 Evolution of Mobile Malware

Android's popularity as a smartphone (mobile device) operating system makes it a popular target for mobile malware writers. Mobile devices can be accessed and exploited via a variety of connections, including mobile networks, WiFi, Emails, Web Browsers, SMS, and MMS. Mobile devices use a variety of technologies, including cameras, accelerometers, Bluetooth, and GPS, all of which are vulnerable via device driver or firmware [30]. Many malware programmes take advantage of a known vulnerability or infect a device via communication endpoints. This section examines the history of mobile malware (Figure 2.4) to identify the recurring vulnerability that the attacker has exploited to infect a device [31].

- `Cabir`, the first mobile malware discovered in the wild, targeted Symbian-based mobile phones in June 2004. The main feature discovered was its ability to use

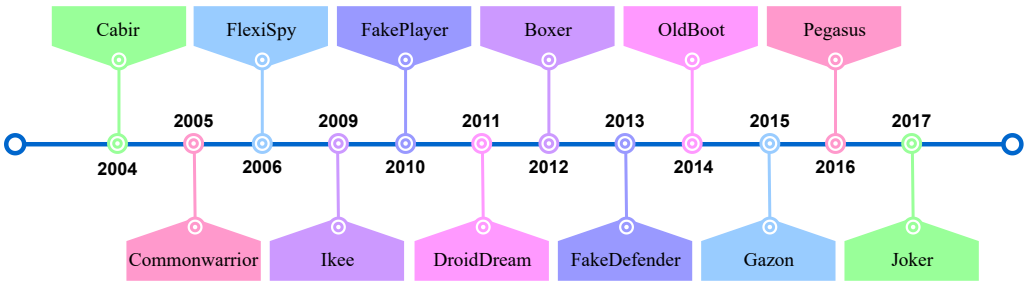



FIGURE 2.4: Timeline of mobile malware

Bluetooth signals to spread to other mobile devices. The source code `Cabir` served as the foundation for a wave of subsequent mobile malware.

- `Commonwarrior`, a computer worm that replicates via MMS and targets Symbian OS, was reported in March 2005. Basic concepts of `Cabir` served as the foundation for `Commonwarrior`. It was the first malware that harmed victims financially.
- In 2006, `FlexiSpy` was born. It was originally created for Symbian OS, but now it is targetting Android and iOS as well. It has the ability to track locations, read messages (SMS and WhatsApp), intercept phone calls, and perform other functions.
- `Ikee`, the first malware for the iOS platform, was discovered in 2009. It spread between jailbroken iPhones using the Secure Shell Connections. Its code was later used to create more malicious iPhone malware known as `Duh`.
- The first Android malware, `FakePlayer`, a Trojan horse, was discovered in August 2010 by the Kaspersky Lab. It took advantage of the SMS service by attempting to send SMS messages to a hardcoded premium number without the user's permission. It spread under the guise of a manually installed movie player app.
- In 2011, the first malware discovered in the official Android market was `DroidDream`. `RootCager` is another name for it. It was a Trojan with two exploit codes: rage-against-the-code and exploid, which were generally used to gain root access on an Android device. It granted the remote attacker root access to the Android device.
- `Boxer`, an SMS Trojan, was discovered in the official Android market in 2012. It was repackaged in several legitimate Android apps. It poses as a legitimate installer

for popular apps such as Skype, Anti-malware, Instagram, and so on. After being installed on a device, it would send an SMS message, prompting the download of a modified app that would continue to send messages to premium numbers.

- In 2013, `FakeDefender` was the first ransomware discovered for the Android operating system. Once installed on the device, the user had to pay a ransom of $99.99 to regain access to the device, which was masked as Android Defender.

- `OldBoot` was the first bootkit designed for the Android operating system in 2014. It had the unusual ability to reinstall itself every time it was uninstalled, making complete removal difficult. During installation, `OldBoot` partially self-installed in the boot partition and changed the initialization scripts that are in charge of installing OS components. This allowed `OldBoot` to run every time a device was turned on.

- `Gazon` malware was discovered in 2015, masquerading as an app that provides Amazon rewards and vouchers worth up to $200. It spreads through SMS messages that promise the recipient a free Amazon gift card. Once the malware is installed on a device, it sends SMS messages to all of the victims' contacts in their contact list.

- `Pegasus` is spyware that was found in August 2016 when a human rights activists' iPhone was tried to be hacked but failed. It was made by the Israeli cyber-arms company NSO Group and can be installed covertly on iOS and Android phones by exploiting zero-day or known vulnerabilities.

- `Joker` malware was discovered for the first time in 2017. It is aimed at mobile phones that run the Android operating system. When `Joker` malware is installed on a phone, it can steal user-specific information and sell it online, generate transactions, and read OTPs in order to conduct financial transactions without the user's knowledge.

## 2.3 Android Malware Analysis Approaches

As statistics show, Android is the main target for malware authors, so Android malware analysis approaches are needed. This section reviews the existing malware analysis approaches, techniques built around them, and their limitations. The malware analysis approaches are classified into three categories: static analysis, dynamic analysis, and hybrid analysis [32]. The description of malware analysis approaches and techniques built by using them are described in the following sub-sections.

### 2.3.1 Static Analysis

We can analyze an app/program in static analysis without executing it. The static analysis process is faster and reveals all possible paths an app can take during execution. However, obfuscation tactics that limit access to the code are threats to all types of static analysis (whether source code or binary). Furthermore, network-related activities and code modification at runtime are often outside the scope of static analysis because they come into the picture during the execution of a program.

In most cases, source code is not available for static analysis of Android malware. As a result, multiple frameworks have been created to reverse engineer the APK and evaluate various components of it (see Section 2.1.2). Androguard [33], Argus-SAF [34] (also known as Amandroid [35]), and APKTool [36] are some of the reverse engineering tools available. These tools have been widely used in static analysis and the development of new malware detection systems.

### 2.3.2 Dynamic Analysis

The dynamic analysis examines an app's behavior while it is running in real-time. This method can discover harmful activities that static analysis cannot detect. However, the dynamic analysis technique has a code coverage problem, which means that some code

parts may not execute during the analysis. Furthermore, by recognizing the underlying emulated platform, platform sensing malware can fool the dynamic analysis process.

Many dynamic analysis frameworks have been developed to examine an Android app dynamically. Droidbox [37], CuckooDroid [38], MobSF [39], and DroidScope [17] are examples of such frameworks. Several of these frameworks have been used to develop malware detection systems. Often such security systems are meant to deploy on the app store to analyze an app offline before distributing it to end-users.

### 2.3.3 Hybrid Analysis

The hybrid analysis takes advantage of the static and dynamic analysis to monitor the malicious behaviour of the app. In this technique, static and dynamic analysis approaches are combined to overcome the limitation of each other.

These are the analysis approaches that a security analyst can use to investigate Android malware. Next, we focus on analysis techniques that use one or more analysis approaches mentioned above to analyze an app.

### 2.3.4 Analysis Techniques

We have seen the analysis approaches that an analyst can use to investigate malware. This section goes through the techniques designed for analyzing an app based on the approaches discussed earlier [32].

#### 2.3.4.1 Network Traffic

The majority of harmful and benign apps require network connectivity with an external entity. For example, a malicious app can only leak sensitive information belonging to a user if it has network access. Otherwise, sensitive data will remain in the device. As a result, network traffic analysis can aid in the detection of malicious activities within a

device. Many studies [6, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52] have proposed to use network information for the detection and categorization of malware. Drebin [6] is a malware detector based on static analysis that employs hardcoded URLs or IPs to determine possible communication endpoints for an app. If the hardcoded URLs or IPs are encrypted, this functionality will negatively influence the performance of Drebin.

Lever et. al [52] uses the Domain name to look at the malicious traffic in cellular carriers. However, not all malware uses domain names to communicate with the server. In general, malware use IP addresses instead of domain names. To test this argument, we took ∼53k samples of malware from 2013 and extracted the communication endpoints (IP address and domain name) from them. After extraction, we found that only about 32% (16,923) of the samples communicate with the external servers and that only about 12.8% (2,162) of the malware samples use domain names to communicate. So, looking only at domain names does not solve the problem of finding the malicious application since most malicious programs either do not communicate with external servers or use IP addresses instead of domain names to communicate with them.

Similarly, many network traffic-based malware detectors like [43, 46, 49, 51] use virtual devices to execute an app to capture network traffic, while some work [40] use real devices. In the case of the virtual device, a platform sensing malware can detect the emulated platform, which triggers malware action not to send any malicious traffic to its peer party. However, the system running on real devices is immune to this attack which is possible by placing a VPN connection to route the traffic through the monitoring entity.

#### 2.3.4.2 Application Programming Interfaces (APIs)

The Android framework provides a diverse range of APIs that allows an app to communicate with the device effortlessly. Some of these APIs allow access to sensitive data critical to some users and are protected with permissions. As a result, APIs can be an excellent source of information for detecting malicious activities, as previously explored [6, 50, 53, 54, 55, 56, 57].

#### 2.3.4.3 System Call

The Kernel in Android provides access to the hardware via system calls. When an app uses a system API to access hardware, the APIs make calls to the system call internally. As a result, system call-centric analysis has been implemented, which tracks system calls' use to determine whether an app is malicious [58, 59, 60, 61].

#### 2.3.4.4 Dependency Graph

The dependency graph depicts the relationship between two statements of the program. If there are two statements, S1 and S2, and S2 is executed after the S1. In this scenario, an edge connecting S1 and S2 demonstrates their inter-dependency. How these edges are created decides the type of a graph. For example, if an edge is drawn to show the invocation of the function one after another, called the function call graph. A dependency graph can be analyzed to find the similarity to detect malicious behavior. Xu et al. [62] utilize the control flow graph (CFG) and data flow graph (DFG) for the classification and characterization of the Android malware family.

#### 2.3.4.5 Information Flow Tracking

In this technique, information of interest is tainted with the tracking information as soon as it is originated and remains in the monitoring state until it reaches the destination (sink). It is generally used to detect leaked data either statically or dynamically. FlowDroid [63] is a static analysis tool to identify information leaks, while TaintDroid [64] and TaintART [65] detect information leaks dynamically. Taint analysis in TaintDroid and TaintART is performed only at the framework level. However, these systems may fail when an information leak happens below the framework layer, i.e., using native code.

#### 2.3.4.6 Opcode

Opcode is a lower-level code found in every software binary. In Android, APK file contains the Dalvik opcode which is generally executed by the Android virtual machine i.e., DVM/ART. The Dalvik opcode can also be used to identify malicious apps. TinyDroid [50] is an example of a tool that detects Android malware using opcode information.

#### 2.3.4.7 Machine Learning Model

Machine learning methods are becoming increasingly popular for detecting and categorizing malware. Many studies [6, 7, 8, 18, 66, 67, 68, 69, 70, 71] have advocated using information from the aforementioned analytical methodologies as a feature and attempting to develop a machine learning model based on past data (datasets) to predict forthcoming malware. Drebin [6] uses static features from application manifest file and Dex code and learn support vector machine to detect malware that may come in the future. Similarly, EC2 [18] employs both static and dynamic characteristics to detect malware families and builds a voting classifier-based ensemble machine learning model. Machine learning based malware detection and classification systems require a dataset with ground truth values to train a machine learning model and evaluate its efficacy.

It has been seen that most of them use an older dataset that does not reflect the current state of the apps. Furthermore, for the testing purpose, they split the dataset randomly to separate testing samples from the training set, which may result in an incorrect time split, means the testing sample might be older than the training set. It might be possible that the performance of these malware detectors degrades when tested on a new sample with realistic conditions. A similar experiment has been done by TESSERACT [72] shows that the performance of Drebin and MaMaDroid [71] dropped by 50%. Therefore, a machine learning based malware detector should be evaluated against the right dataset, free from all the experimental biases.

## 2.4 Countering the Malware Analysis Process

Because malware writers are aware of the malware analysis process, they are constantly looking for novel ways to disguise malware from detection. For this purpose, many methods/techniques have been created to counter the malware analysis process. This section provides an overview of counter analysis methods in the context of static and virtual device-based dynamic analysis processes.

### 2.4.1 Obfuscation Techniques

Obfuscation is a popular way to avoid static analysis. There are two types of static obfuscation techniques: trivial and non-trivial. In the following part, we examine both obfuscation methods.

#### 2.4.1.1 Trivial Obfuscation

These are the most straightforward techniques that do not have a real obfuscation effect on an APK. However, they can fool signature-based malware detectors. This obfuscation technique transforms the APK file structure without altering its semantics. However, it does not deal with the code of the APK. In simpler words, the hash of the resulting APK file differs while preserving the same original functionality of the App. The possible obfuscations in this category are as follows:

***(i)*** **Re-align:** Aligning an APK file entails improving its structure in order to efficiently map the final archive in memory. Aligning an archive has the unintended consequence of creating a slightly different file with the same functionality as the original.

***(ii)*** **Re-signing:** A digital certificate must be used to sign an Android app so that its integrity can be checked during installation. When we re-sign an app with a different certificate, we get a new file with a different hash than the original, but the functionality

remains the same. Some anti-malware programs cannot detect transformed malware that uses message digest as a detection mechanism.

***(iii)* Repackaging:** As previously stated, an Android app is a compressed file similar to a jar file. Suppose we uncompress the APK file and add a new dummy or empty file, followed by repackaging. This process generates a new APK with a different structure and message digest.

***(iv)* Randomize Manifest:** Rearranging the Android Manifest file entries without modifying the XML tree structure results in a different hash of the Manifest file. As a result, we get a new APK file while preserving the original functionality of the app.

#### 2.4.1.2 Non-trivial Obfuscation

Non-trivial approaches are more difficult to implement, but they provide a better return in terms of detection rate and robustness. This method targets both the bytecode and the resources of an APK. Renaming, Encryption, Code, and Reflection are the four subcategories of non-trivial obfuscation techniques. The following are the details of these categories:

***(i)* Renaming:** The names of classes, functions, fields, activities, packages, and a variety of other resources can all be utilized to detect malware on Android. For example, one of the elements used by Drebin [6] is the name of activities, services, content providers, and others. Drebin can fail to detect malware if this information is renamed in an APK. The renaming obfuscation technique can be used to obfuscate both code and non-code resources.

***(ii)* Encryption:** AV signatures are mostly based on the sequence of bytes acquired from an app's various resources. Native libraries, data, strings, and assets files are examples of resources that an app can use at runtime. An app can encrypt and decrypt such resources at execution time in encryption-based obfuscation. This obfuscation technique is not only

limited to fooling signature-based AV products, but it can also fool flow-based and ML-based detectors.

***(iii)* Code:** This obfuscation technique mainly affects the instruction code inside the Dex files. It is the most advanced obfuscation techniques that alters the Dex file by using one or more different methods listed below:

```
Junk code insertion
Removal of debug symbols
Call indirection
Adding new method by exploiting method overloading
Adding Nop instruction
Reordering of code
Unconditional jump
Dummy Arithmetic and Branch instruction
```

This obfuscation technique targets all types of static malware detection techniques as discussed earlier.

***(iv)* Reflection:** It is a process of examining and changing a class's runtime behavior during execution time. In reflection-based obfuscation, the invocation of a function is diverted to a bespoke function that calls the original function. Most static analysis tools often fail to capture such behavior because the actual function invocation is determined at execution time.

### 2.4.2 Dynamic Code Loading

It is a method that allows an app to load a binary or library during execution. This binary or library can be saved as an asset inside the app or can be an external source that an app downloads during runtime. In any case, it is not visible to a static analysis tool because of its dynamic nature, which eventually fails the analysis process [32].

### 2.4.3 Packed Malware

To hide a malicious behavior from being caught by the static analysis tool, malware writers encrypt (packed) the entire code except for the unpacking logic. First, this type of malware at runtime unpacks the encrypted code. Later, they use dynamic code loading techniques to execute this unpacked code to perform its designated task. As the malicious behavior of the app is stored in the form of encrypted code, none of the static analysis tools can reveal its malicious nature [73].

### 2.4.4 Platform Sensing Malware (Emulation-Detection)

The primary issue with an emulated system is its inability to replicate a complete system that matches the exact configuration and characteristics of a physical device. The core idea of emulation-detection is to observe the differences between virtual and physical machines using a program to identify the underlying infrastructure. Vidas et al. [74] and Morpheus [75] have shown that such differences can be used to detect underlying emulated platforms through a stand-alone app. Vidas et al. [74] propose a few generic detection methods based on the device characteristics, e.g., differences in hardware components (like sensors and CPU information) and software components (like Google apps are not present). Morpheus [75] presents more than 10000 heuristics to detect the underlying emulated platform has broadly classified it into three categories viz. i) Files, ii) APIs, and iii) System Properties related detection which are similar to the techniques proposed in [74]. The emulation-detection methods shown in [74, 75] fall in the category of basic emulation-detection, and most of the dynamic analysis systems are capable of bypassing them.

## 2.5 Observations

As previously stated, Android is becoming the most popular target for malware writers because it is open source and has the largest share of the global smartphone market. We have studied Android malware analysis and detection technologies to analyze their performance to grasp the current malware threat better. This section presents our findings from this study, which will serve as the foundation for this thesis.

### 2.5.1 Anti-Emulation-Detection Capabilities

Offline security system deployed on app stores heavily relies on the emulated platform for dynamic analysis. Platform sensing malware can easily bypass the dynamic analysis process by detecting the underlying emulated platform and subsequently entering into the mobile device. For example, Google Play Protect, which is used to certify Android apps, fails to detect malware that spreads across 85 different apps affecting nine million Android devices [20].

We believe, a dynamic analysis system should provide a configurable anti-emulation-detection mechanism, so that a smart malware developer could find it difficult to evade the dynamic analysis by studying the analysis framework. Moreover, we would like to emphasize the need for a validation mechanism to understand the anti-emulation-detection measures incorporated by a dynamic analysis framework.

### 2.5.2 Cross-layer Profiling

Android apps can be developed using Java and native code, which crosses the layers of Android. Most dynamic analysis frameworks profile an app at the framework layer (i.e., only a single layer). A malware writer can split the malicious behavior of an app across multiple layers. In that case, the existing dynamic analysis framework can fail to capture the malicious behavior of such malware. Therefore, a dynamic analysis framework

should provide the capability to profile an app across multiple layers while introducing low profiling overhead.

### 2.5.3 On-device Malware Detection

Malware may get unleashed into the device, bypassing the defense system of Google Play Store or from unverified sources (e.g., third-party market, sideloading). As a result, malware detection on real devices is critical for preventing malware infection. One option is to install the existing offline system on the device. However, these systems require high resources and processing time, which can reduce the device's battery life. Hence, there is a requirement for a lightweight malware detection system that we can deploy on resource constraint mobile devices.

### 2.5.4 A Lack of Representative Dataset

As discussed earlier, existing datasets used to test how well machine learning-based malware detectors work are old and do not reflect the current state of malware. The efficiency of malware detectors evaluated against these datasets is questionable due to the lack of a representative dataset. Therefore, we focus on creating the right and biased free dataset that depicts the state of current malware.

### 2.5.5 Focus on Family Identification

Malware analyst has given a minor focus on automatic identification of malware family, which is crucial to identify the damages caused by malware and how to get rid of it. Therefore, we also put our focus on the identification of malware families.

## 2.6 Summary

This chapter has covered the basics of Android OS and its components. We next went over multiple malware analysis and detection approaches for Android. We have also looked at how mobile malware evolves to evade detection and obfuscate analysis. By analyzing both malware threats and solutions, we have made several observations that are the key points for this thesis work. From here onwards, we concentrate on our observations, starting with creating tools and datasets that we use to assess the efficacy of an analysis framework.

# Chapter 3

# Datasets and Tools

Some tools and a dataset with ground truth values are required to construct and assess a malware detection system. In general, tools are used to extract relevant information from a sample to discover harmful activity in an app, if any. The dataset, on the other hand, aids in the evaluation of a malware detection system's performance. The tools and libraries that have been utilized to extract important information to construct an effective malware detection system are introduced in this chapter. We then go over the datasets we use and where they came from.

## 3.1 Tools and Libraries

This section outlines the tools and libraries we employ to design an effective malware detection system, whether we create them ourselves or use existing ones.

### 3.1.1 Emulation Detection Library (EmuDetLib)

To study the effectiveness of the existing dynamic analysis frameworks, we require a tool with varying levels of the emulation-detection method. We have developed a flexible **emu**lation-**det**ection **lib**rary (EmuDetLib). The detection techniques in this library can be

broadly classified into five categories (refer Table 3.1): *(i)* Unique device information, *(ii)* Sensors' reading, *(iii)* Device state information, *(iv)* GPS information, and *(v)* Distributed detection.

Table 3.1: Classification of emulation-detection techniques.

| **Detection Categories** | **Description** |
|---|---|
| Unique device information (basic) | Detection by observing unrealistic device information values (e.g., IMEI value is 00000) |
| Unique device information (smart) | Detection based on fixed reading of unique device information (e.g., IMEI value is constant) |
| Sensors' reading | Absence of sensor or observing static values from fluctuating sensors (e.g., fixed reading of `Light` sensor) |
| Device State information | No change to the device state w.r.t. telephony signal, battery power. |
| GPS information | No change on GPS location data or fake location change |
| Distributed detection | Observing identical unique information for multiple devices in a network. |

#### 3.1.1.1 Unique Device Information

As many Android devices are smartphones, Android provides a comprehensive API that includes the interfaces to query telephony information along with unique data per device like IMEI, IMSI, and phone number. These details may help in identifying a device uniquely. For example, on querying Device ID (IMEI) from an emulator, it returns all '0', which is unrealistic and indicates that the device is an emulator. Similarly, other information related to a SIM card like SIM serial number, phone number, and IMSI can reveal the identity of the underlying platform, i.e., whether the platform is emulated or not.

To hide the emulated platform against the emulation-detection based on the unique device information, a sandbox designer can supply more realistic data when queried by an app. Most of the dynamic analysis framework such as CuckooDroid, Droidbox, and MobSF provides similar anti-emulation-detection measures by feeding-in realistic device information. However, a smart malware author can analyze this data and find that they are static (fixed) to successfully detect the underlying platform.

#### 3.1.1.2 Sensors' Reading

Today's smartphones have various sensors for different purposes that can be broadly classified into two categories—motion sensors and environmental sensors. The motion sensors help to detect the motion on a device. Such sensors can measure the acceleration, magnetic field, rotation, and others along the three dimensions, i.e., X, Y, and Z, which indicates that the smart-phone user has made some motion. Similarly, the environmental sensors observe the changes in the operating environment so that an app can alter its behavior based on the readings. For example, a light sensor provides the luminosity information of the environment which can be used by a smartphone to adjust the display brightness.

For an Android app, the use of these sensors do not require any permission and an app can take full advantage of these sensors. As the data observed on these sensors fluctuate continuously, this insight can be used to detect the underlying emulated environment. A recent example of sensor-based detection is the observation of TrendMicro where malware (in Play Store) make use of the motion-detection feature to evade the dynamic analysis [76]. This is an indicator that the dynamic analysis framework on the play store is not secure against the advanced emulation-detection techniques.

Identifying such sophisticated malware is not trivial; hence, we have incorporated sensor-based detection functionalities in our library for emulation-detection. Our library provides the detection mechanism based on the motion sensors as well as on the environmental sensors.

#### 3.1.1.3 Device State Information

In reality, a device state gets changed due to some internal/external events such as changes in telephony signal strength, battery power, and incoming SMSes/Calls. However, such state changes behavior is missing in an emulated environment. Therefore, information related to a device state can also be queried by malware to detect the emulated environment as described below:

**(i) Telephony signal:** As the signal strength of the telephony system cannot remain constant for a long time, malware can observe such information over a period. The malware can flag the underlying operating environment as emulated if it finds that the signal strength is constant.

**(ii) Battery power:** Similar to the telephony signal, the batterys' power strength which is the standard behavior of every mobile device. If such behavior is absent, and battery power remains constant, it indicates that the operational machine is virtual.

**(iii) Incoming SMS/call:** A malware can register a notification for every incoming SMS/Call, and wait for the event. In an emulated environment, malware may not observe such an event and will thus remain dormant without performing any malicious activities.

#### 3.1.1.4 Using the GPS Location Information

GPS (Global Positioning System) is also a sensor and the malware can use similar methods (as explained above) that are used for other sensors to detect emulated platforms. However, the emulation-detection based on the GPS is somewhat different from other sensors, as explained below.

Android provides rich APIs to perform various tasks, one such API gives the power to generate a mock location that can be used by an app to introduce a fake location when queried. An Android app requires `ACCESS_MOCK_LOCATION` permission to use the mock location API. The other source for geo-location is BTS ID. Android provides API to query BTS ID, and we can get its geo-location by using publicly/commercially available services (`https://opencellid.org`). Hence, the geo-location-based emulation-detection technique only works when one of the following conditions is satisfied: *(i)* there is no change in the geo-location of the device, *(ii)* the mock location API is used to set the geo-location of the device, or *(iii)* BTS geo-location is not collaborating with the GPS location.

Detection in case of condition *(i)* is easy; since we do not see any changes in geo-location, we can directly infer that a device is an emulator. However, exploiting *(ii)* or *(iii)* condition is non-trivial as fluctuating GPS data can be a result of fake location app or real change in location. Detecting emulation in the *(ii)* condition requires a mechanism to differentiate between real location data and mock location data. In contrast, the *(iii)* condition assumes that the distance between GPS location and BTS geo-location can not be more than $x$ meters. The value of $x$ may differ depending on the density of the area where BTS is located.

The `Location` module of the Android framework provides explicit methods to differentiate the location data w.r.t. its source i.e. whether the data is a mock location or a real GPS device reading. Malware can use this method to detect emulation platforms. However, this check may fail if the mock location generation app is either closed or running as a service. In this scenario, the `Location` module returns the GPS data (probably set by a fake location app executing before) without the fake location indicators. Therefore, the mock location detection feature provided by the `Location` module doesn't help and requires a more fine-grained mechanism to differentiate mock locations from real location data.

Another possible approach to be sure about the location data being fake is to identify active apps that can generate fake location information. As discussed earlier, a fake location generation app requires special `ACCESS_MOCK_LOCATION` permission. Upon querying the package manager, we can get a list of the apps and services that have been granted this permission and subsequently checking them against the list of running processes. If we find a running app or service, then we can say that the current location is most probably fake and can consider it to be an emulated environment.

Algorithm 1 detects such apps and returns `true` or `false` based on the status of the fake location apps currently executing on the device (if any). The algorithm queries the Android package manager for the permissions granted to individual apps, and identifies

**Algorithm 1:** Fake Location App Detector

$returnValue \leftarrow false$
$RunningApp_{pkgName} \leftarrow \phi$
$RunningApps_{process} \leftarrow getRunningAppProcesses()$
**foreach** $App_{process}$ *in* $RunningApps_{process}$ **do**
    $PackageName \leftarrow getPackageName(App_{process})$
    $append(RunningApp_{pkgName}, PackageName)$
$RunningServices \leftarrow getRunningServices()$
**foreach** *service in RunningServices* **do**
    $PackageName \leftarrow getPackageName(service)$
    $append(RunningApp_{pkgName}, PackageName)$
**for** $i \leftarrow sizeof(RunningApp_{pkgName})$ **to** 1 **do**
    $PackageName = getItemAt(RunningApp_{pkgName}, i)$
    **if** *not* $hasMockLocationPermission(PackageName)$ **then**
        $removeItemAt(RunningApp_{pkgName}, i)$
**if** $sizeof(RunningApp_{pkgName}) > 0$ **then** `// Checks if there is any process running with MOCK location permission`
    $returnValue \leftarrow true$
**return** $returnValue$

those with the `ACCESS_MOCK_LOCATION` permission. If at least one app/service is active with the permission, the algorithm returns `true`.

#### 3.1.1.5 Distributed Emulation-detection

So far, we have discussed the emulation-detection techniques that can be employed by a standalone app to detect the emulated environment. Nowadays, most apps require communication with a centralized server to share their status or get new information. To identify a device uniquely at the server, an app typically generates a unique ID called an AppID. Apart from the AppID, a smartphone also contains device-related unique IDs namely IMEI, IMSI, SIM Serial number, and others. This information can also help in identifying a device uniquely at the server as explained below.

It is trivial to see that a slightly different malware in terms of its signature can be generated easily by changing its package name, altering the function name and variable naming convention, or by introducing dummy code while retaining the overall functionality and the server address. Such malware can communicate the unique device information to

a remote server to identify the emulated environment remotely. In this situation, the emulation-detection can happen at the server by querying the device information from the connected devices. If a server detects that multiple devices have identical information (expected to be unique), it can flag those devices as emulated environment. As this emulation-detection is carried out in the context of multiple connected devices, we classify this detection technique as a distributed emulation-detection.

To detect emulation using the distributed emulation-detection mechanism, we need a server where the emulation-detection mechanism is deployed. A candidate emulation-detection procedure that can be deployed on the centralized server is described in Algorithm 2. As shown in the emulation-detection algorithm, the devices identified as an emulated platform are added to the blocked device list. The server can execute this detection process either on a new device connection request or on every request received from the client. On detection of the emulated device, the server can either stop serving the client or can notify the client about the emulation-detection.

**Algorithm 2:** Blocking Emulated Device (server)

$Devices_{blkd} \leftarrow getBlockedDevices()$
$Devices_{con} \leftarrow getConnectedDevices()$
$Properties \leftarrow \{IMEI, IMSI, ....\}$ // Unique device information
$Devices_{emu} \leftarrow \phi$
**foreach** *Prop in Properties* **do**
    $Devices \leftarrow Devices_{blkd} \cap_{prop} Devices_{con}$
    $Devices_{emu} \leftarrow Devices_{emu} \cup Devices$
**foreach** *Prop in Properties* **do**
    $Devices \leftarrow groupByProperty(Devices_{con}, Prop)_{count>1}$
    $Devices_{emu} \leftarrow Devices_{emu} \cup Devices$
**foreach** *device in* $Devices_{emu}$ **do**
    $addToBlockedDevice(device)$

#### 3.1.1.6 Ethical Concern

EmuDetLib is designed for malware analysts, vulnerability assessment and penetration testing (VAPT) engineers, and in general, cybersecurity researchers who are interested in

checking vulnerabilities in emulator-driven dynamic analysis framework. However, there is a concern that such a library might also be misused by unethical hackers for finding vulnerabilities in already released popular analysis frameworks and use the knowledge to bypass the dynamic analysis process. We also note that this ethical concern is often present in all research publications that presents a VAPT tool or disclose a vulnerability in products.

### 3.1.2 Obfuscapk

Obfuscapk [3] is a Python-based black-box obfuscation tool for Android apps that do not require source code. Ofuscapk employs APKTool to decompile an APK. It then uses obfuscation techniques on the decompiled smali code, manifest file, and other resources to make a new app. The obfuscated app still works the same way as the original app, but the code changes can make the new app look different from the original. Obfusapk includes advanced obfuscation features divided into five categories. Table 3.2 lists the possible obfuscators implemented in Obfuscapk's various obfuscation categories. We use this tool to create a new obfuscated dataset to evaluate the efficacy of static Android malware detectors.

TABLE 3.2: Obfuscators implemented in Obfuscapk [3] tool.

| **Category** | **Obfuscators** |
|---|---|
| trivial | Randomize manifest file, rebuild, new alignment, re-signing |
| renaming | Renaming the class, fields and methods |
| encryption | Encryption of library, resource strings, assets, and constant strings |
| reflection | Invoke user defined and framework APIs using the reflection APIs |
| code | Junk code insertion, instruction re-ordering, calls redirection, removing debug data, insertion of goto instruction, adding new method by exploiting method overloading. |

### 3.1.3 Androguard

Androguard [33] is a reverse engineering tool for Android apps that is written in Python. It takes raw APK files of an app and disassembles them for analysis. Androguard is commonly used to perform malware penetration testing and to identify vulnerabilities in Android apps.

### 3.1.4 DexLib2

*DexLib2* [77] is a Java library for processing Dalvik executable code, which is used by several APK processing frameworks, such as APKTool, to undertake reverse engineering. *DexLib2* makes use of smali/backsmali, an assembler/disassembler for the Dex file format used by DVM/ART. On an actual mobile device, we use *DexLib2* to extract the functionality.

## 3.2 Datasets

A dataset with ground truth values is necessary to measure the malware detection system's effectiveness. There are a plethora of such datasets that have been utilized in the past work. Malgenome [78], Drebin [6], Praguard [79], and AMD [80] datasets fall within this category. The Malgenome and Drebin datasets contain outdated malware samples as of October 2012. As a result, these datasets are insufficient to evaluate a malware detection system. Aside from these datasets, there are a number of online repositories where the most recent malware samples can be found. VirusShare.com [81] and the AndroZoo [82] project are two such repositories.

VirusShare.com repository contains live malware samples that they provide to security researchers, incident responders, and forensic analysts. It contains malware samples for all platforms, including Windows, Linux, and Android.

The AndroZoo project is a growing collection of both good and bad Android apps that can be used to study Android malware. It has more than 19 million apps that have been marked as malware or benign by more than one AV engine.

Aside from these online repositories, the Google Play Store, the official Android apps market, can also be considered a source of Android samples. We use samples from VirusShare.com, AMD dataset, AndroZoo project, and Google Play Store in our work. We created a number of datasets to test the malware detection system. Malware samples

were obtained from VirusShare.com, AMD, and AndroZoo, while benign samples were obtained from Google Play and AndroZoo. We also use Pegasus malware samples to evaluate the efficacy of malware detection system. Subsections following this one describes the created multiple datasets.

### 3.2.1 D1:Base-Dataset (2012-2018)

This is a class balanced dataset consisting of 96,748 apps. The dataset contains 40,402 unique malware samples spread across more than 70 different families. This distributed set of malware assists the classifier in learning more about different variants in order to identify previously unseen malicious apps. Malware samples were collected from AMD and VirusShare.com for inclusion in the dataset. We crawled the Google Play Store for benign apps and collected 65,806 samples. These samples were submitted to VirusTotal.com to ensure that they were benign. Apps that have been identified as malicious by even one antivirus engine on VirusTotal are discarded, yielding 56,346 benign samples. The AMD dataset contains 24,553 malware samples collected between 2012 and 2016, distributed across 71 families, whereas the malware from VirusShare and benign apps from Google Play were collected in April 2018. Table 3.3 displays the statistics of malware and benign samples collected from various sources.

TABLE 3.3: Number of malware and benign samples collected from different sources for base dataset D1:2012-2018, where hyphen (–) denotes that samples are not available.

| **Source** | **Malware** | **Benign** | **Year-Range** |
|---|---|---|---|
| AMD [80] | 24,553 | – | 2012 – 2016 |
| VirusShare [81] | 20,979 | – | Till April 2018 |
| Play Store | – | 56,346 | Till April 2018 |
| Total Samples | 45,532 | 56,346 | 2012 – 2018 |
| Unique Samples | 40,402 | 56,346 | 2012 – 2018 |

### 3.2.2 D2:AndroZoo-2019

It is also a class-balanced dataset with 10,760 distinct samples. It contains 5,380 malware samples, with the remainder being benign. Both the malware and benign samples were obtained from the AndroZoo project, where the Dex file's last modification date is of

2019. We consider samples from this dataset to be new samples in comparison to the Base-Dataset because they were born after the Base-Dataset.

### 3.2.3 D3:Pegasus

Pegasus malware is currently in the spotlight in all countries due to its use in mass surveillance to gather intelligence. As a result, we use the Pegasus samples to assess the efficacy of malware detection systems. This dataset includes five samples obtained from the CloudSek organisation. The collected Pegasus malware samples are of earlier than 2019.

### 3.2.4 D4:Obfuscated

Performance of malware detectors degrades in the presence of obfuscated malware samples. Most of the malware detectors like DroidSieve use the PRAGuard [79] dataset to evaluate their efficiency against obfuscated malware. PRAGuard dataset contains malware till March 2013. These samples are outdated and do not represent the current state of apps. Also there may be overlapping samples with our Base-Dataset. Therefore, to avoid inclusion of known obfuscated malware samples in the training data, we created a new obfuscated malware dataset by obfuscating the malware samples of D2:AndroZoo-2019 dataset (5,380 samples of year 2019). To obfuscate malware samples, we have utilized the Obfuscapk tool that can obfuscate sample with five obfuscation techniques (see Section 3.1.2). Out of the 5,380 malware samples, we have successfully obtained 4,993 obfuscated samples in six categories. Five categories are the same as provided by the Obfuscapk, whereas the sixth category comprises a mix of two or more obfuscation techniques (referred to as mix). The number of obfuscated samples in each category are shown in Table 3.4.

TABLE 3.4: Category-wise obfuscated malware samples.

| Category | #Samples |
|---|---|
| trivial | 160 |
| renaming | 570 |
| encryption | 1,135 |
| reflection | 252 |
| code | 2,429 |
| mix | 447 |
| **Total** | **4,993** |

### 3.2.5 D5:Biases-Free

Many Android malware detectors ([71, 7, 6, 83, 84]) are available that publish high detection results of up to 99%. However, there are two potential experimental biases in the experiment (shown by TESSERACT [72])–*(i)* Spatial bias and *(ii)* Temporal bias. Spatial bias occurs due to the incorrect distribution of malware and benign samples in the dataset, whereas temporal bias refers to the incorrect time splits of training and testing samples. Therefore, to evaluate appropriately against potential biases, we have downloaded 87,632 (with ∼10% malware) unique samples from AndroZoo spanning over four years (2016 to 2019). In this dataset, each quarter of every year contains ∼10% malware, and the remaining are benign. Table 3.5 shows the quarter-wise statistics of the dataset.

TABLE 3.5: Quarter-wise statistics of the Biases-Free dataset.

| Year | Quarter | #Malware | #Benign | Total |
|---|---|---|---|---|
| 2016 | Q1 | 1,885 | 16,968 | **18,853** |
| | Q2 | 1,263 | 11,365 | **12,628** |
| | Q3 | 1,354 | 12,185 | **13,539** |
| | Q4 | 608 | 5,470 | **6,078** |
| 2017 | Q1 | 562 | 5,007 | **5,569** |
| | Q2 | 340 | 3,001 | **3,341** |
| | Q3 | 251 | 2,224 | **2,475** |
| | Q4 | 271 | 2,392 | **2,663** |
| 2018 | Q1 | 240 | 2,120 | **2,360** |
| | Q2 | 259 | 2,301 | **2,560** |
| | Q3 | 363 | 3,210 | **3,573** |
| | Q4 | 320 | 2,813 | **3,133** |
| 2019 | Q1 | 237 | 2,088 | **2,325** |
| | Q2 | 190 | 1,696 | **1,886** |
| | Q3 | 314 | 2,786 | **3,100** |
| | Q4 | 362 | 3,187 | **3,549** |
| **Overall** | | **8,819** | **78,813** | **87,632** |

## 3.3 Summary

This chapter has introduced an Emulation-Detection library EmuDetLib. Emulation-detection capabilities in EmuDetLib are configurable and can easily integrate with any app to measure the anti-emulation-detection capability opted by any dynamic analysis framework. Later, we created multiple datasets to evaluate the efficacy of machine learning based malware detectors in all possible scenarios. Table 3.6 shows the summary of the created datasets.

TABLE 3.6: Datasets Summary

| **Dataset** | **#Malware** | **#Benign** | **Total** | **Year-Range** |
|---|---|---|---|---|
| D1:Base-Dataset | 40402 | 56346 | 96748 | 2012 – 2018 |
| D2:AndroZoo-2019 | 5,380 | 5,380 | 10,760 | 2019 |
| D3:Pegasus | 5 | – | 5 | before 2019 |
| D4:Obfuscated | 4,993 | – | 4,993 | 2019 |
| D5:Biases-Free | 8,819 | 78,813 | 87,632 | 2016 – 2019 |

In the next chapter, we empirically evaluate the anti-emulation-detection capabilities of the well-known dynamic analysis framework. Later we design a stealthy dynamic analysis framework using the insights learned from the empirical evaluation.

# Chapter 4

# InviSeal: A Stealthy Dynamic Analysis Framework for Android Systems

With wide adaptation of open-source Android into mobile devices, sophisticated malware are developed to exploit security vulnerabilities. As comprehensive security analysis on physical devices is impractical and costly, emulator-driven security analysis has gained popularity in recent times. Existing virtual device-based dynamic analysis frameworks suffer from two major issues. *(i)* they do not provide foolproof anit-emulation-detection measures, and *(ii)* lack efficient cross-layer profiling capabilities. This chapter presents InviSeal, a comprehensive and scalable dynamic analysis framework that includes low-overhead cross-layer profiling techniques and detailed anti-emulation-detection measures along with the basic emulation features. Firstly, we empirically evaluate the anti-emulation-detection capabilities of existing dynamic analysis frameworks. Secondly, we design a stealthy dynamic analysis framework InviSeal using the insights learned from the empirical evaluation. Lastly, we evaluate the efficacy of InviSeal, followed by some usage scenarios.

## 4.1 Introduction

Existing offline analysis methods that deal with security issues caused by the rapid growth of Android malware can be roughly put into two groups: static analysis and dynamic analysis/detection techniques [19, 85]. As discussed earlier, techniques based on only static analysis [8, 33, 34, 35] are insufficient to address the security issues presented by the malware especially designed to bypass the static analysis based defenses. For example, advanced malware employ techniques such as dynamic code loading, native code exploitation, Java-reflection mechanisms, and code encryption to bypass the static analysis based detection [32, 86]. Dynamic analysis and detection techniques are the most commonly used and researched in the contemporary mobile security arena and is the scope of this chapter.

Existing dynamic analysis and detection techniques address specific security issues like information leakage through device channels, untrusted mobile activities through different apps in a secure environment. For example, dynamic information flow tracking (DIFT) based malware detection techniques like taintDroid [64] and taintART [65] track the information flow from the source (sensitive information) to sink (device channel), to detect information leakage while behavior graph based techniques [87] identify potential malware based on call graphs. Other techniques are based on network traffic monitoring [88] and cloud usage monitoring [89] to detect potential malware. However, all of the above techniques are limited to specific security risks and do not provide a comprehensive platform for malware analysis and detection. For example, taintART [65] provides information tracking only at the framework layer that cannot detect malware leaking information through native code. Moreover, techniques like taintART when used on a real device *result in significant performance overheads ($\sim$10%) and increased energy usage.*

Generic sandbox based techniques (Droidbox [37], CuckooDroid [38], DroidScope [90], MobSF [39]) provides a sandboxed virtual environment to perform malware analysis and detection by executing apps inside the sandboxes and collecting various event logs.

**The problem:** Even though a wide variety of Android sandboxes are available for app analysis, malware can bypass the dynamic analysis process running on these frameworks by employing one or more techniques listed below.

***(i)*** Many malware [74] employ techniques to detect the underlying emulation platform before showing their true behavior. As far as we know, none of the existing emulator driven dynamic analysis frameworks make claims regarding their effectiveness towards nullifying possible emulation-detection adopted by malware. For example, CuckooDroid [38] provides fixed data when different device specific information (e.g., GPS and IMEI) is queried from the app which allows a malware to observe them and evade the dynamic analysis defenses [91].

***(ii)*** CuckooDroid and taintART based sandboxes depend on the profiling information collected from the framework layer, therefore *cannot detect information leakage through the native code and the OS level system call APIs.* For example, two colluding malicious apps can use system calls (like `mmap()`) to setup a shared memory communication channel without being detected by the above techniques.

***(iii)*** `Strace` [92] is a widely used utility to profile system call information to enable cross layer profiling. However, `strace` incurs high overhead and slows down the app execution, opening another way to detect the monitoring environment. Also, `strace` based profiling system can be detected from the malware using a simple detection method where the malware launches `strace` on itself. In this scenario, given that the `strace` based dynamic analysis system is already tracing the malware app, the `strace` instance launched by the malware will fail which will expose the dynamic analysis process.

**Our goal:** Ideally a desirable mobile emulator platform for security analysis should provide the following features: *(i)* it must have cross-layer (application layer to OS layer) profiling capabilities, *(ii)* built-in anti-emulation-detection measures for robust malware analysis, and *(iii)* incur low profiling overheads. Apart from the above mentioned features, memory dump and packet capture features should also be supported which may be used if required for offline analysis.

**Our proposal:** In this chapter, we present InviSeal, a stealthy, low-overhead and comprehensive dynamic analysis framework for apps in the ART execution environment. Firstly, we develop **S**ensor, **T**elephony system, and **D**evice state information **Neut**ralizer (STDNeut) to bypass the sensors based emulation-detection strategy that is fully designed using Qemu [93] based Android emulator [94]. Secondly, we design ARTmon using `Droidmon` module (based on `Xposed` framework) to bypass file and system properties based emulation-detection along with the ability to instrument framework level API. Furthermore, we develop an OS-level system call interposition technique to profile the system call activities in an efficient manner. We provide a one-place log storage system for further analysis and detection of malware. Overall, our contributions are as follows:

***(i)*** We design STDNeut that remain undetected even if the emulation-detection is performed at any layer of the Android OS w.r.t. sensors, telephony system and device state (Section 4.4).

***(ii)*** We modify the `Droidmon` module to support file and system properties based anti-emulation-detection (referred to as ARTmon, Section 4.5). We develop SysCallMon (Section 4.6), a configurable low-overhead OS-level utility to monitor system calls invoked from apps.

***(iii)*** We propose a comprehensive Android security audit framework based on the above utilities that provides a single-point dynamic profiling and analysis support (Section 4.7).

***(iv)*** Our evaluation of InviSeal shows that on an average the profiling overheads is ∼1.04X, which is better than contemporary techniques. Moreover, SysCallMon is ∼1.26X faster than `strace`-based system call profiling. Further, we show the benefits of *cross-layer profiling*, *anti-emulation-detection measures*, and other features in practical usage scenarios (Section 4.8).

## 4.2 Relevant Background

In this Section, we explain the working of `Xposed` framework and `Droidmon` module that are the base for some module to design robust dynamic analysis framework.

### 4.2.1 Xposed Framework

`Xposed` [95] is a widely used generic hook framework for the Android platform. To work appropriately, `Xposed` requires a rooted device. It has mainly four components: `Xposed`, `XposedBridge`, `XposedInstaller` [96], and `XposedMods`. `Xposed` and `XposedBridge` provides support to accommodate the hook framework inside an app. `XposedInstallers` main responsibility is to manage the `Xposed` framework and modules developed by other developers on top of `Xposed`. `XposedMods` are the modules designed to perform a specific task such as changing the behavior of an existing device. Figure 4.1 illustrates the workflow of the `Xposed` framework.

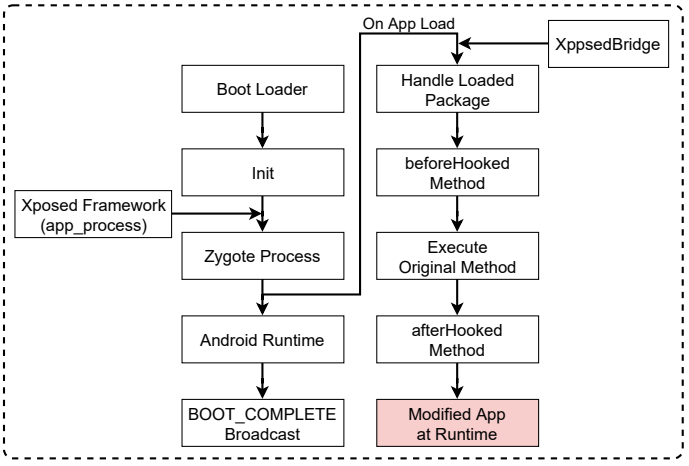



FIGURE 4.1: Workflow of Xposed hook framework.

When an Android phone starts, boot-loader calls the kernel, which loads the first process called `init` at the end of OS boot. `Init` process is responsible for setting up all the daemon processes and components of the Android platform. After the successful start of the `init` process, it invokes the core Android component called Zygote, which is responsible for loading other apps. In a `Xposed`-hooked system, a modified Zygote process is loaded by the

`Xposed` framework to hijack the overall control of the system. After this step, whenever any app is executed, the modified Zygote loads the app by setting up the execution environment for them. At the beginning of the app execution, `Xposed` loads `XposedBridge` library as part of the app executable. The `XposedBridge` library provides necessary API used by the `Xposed` module to change the app behavior. An `Xposed` module registers the API to be hooked with the `Xposed` framework along with the handler methods, invoked before and after a hooked API executes. As a result, `Xposed` module can be used to change the behavior of any app at run time.

Other `Xposed` like hook-based frameworks such as ARTist [97] and ARTDroid [98] are also proposed. However, these frameworks are not as popular as `Xposed` as they lack necessary software support.

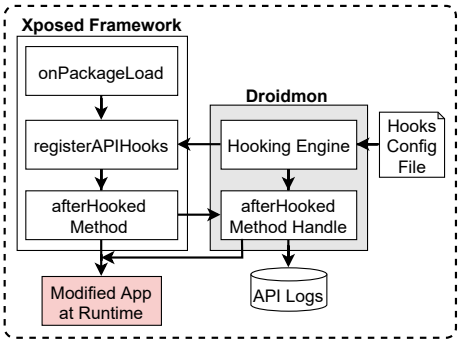



FIGURE 4.2: Droidmon design using Xposed framework hooks.

#### 4.2.1.1 Droidmon

`Droidmon` [99] is an `Xposed` module designed for monitoring the framework level API used by apps in CuckooDroid [38] in a configurable manner. Figure 4.2 shows the design of `Droidmon` module leveraging the hook support provided by `Xposed` framework. When an application package loads, hook engine of `Droidmon` parses the hooks configuration file and provides the details of the API to the `Xposed` framework for instrumentation. The `Xposed` framework (explained before) enables hooks at different app execution points. `Droidmon` primarily makes use of the `afterHooked` method (refer Figure 4.1 to record information related to the invoked API (e.g., return value and arguments). `Droidmon` is used to monitor

the API in a non-intrusive manner without modifying the original app methods during the monitoring phase.

## 4.3 Motivation

This Section motivates our proposal and discusses the limitations of the existing sandbox-based profiling techniques.

### 4.3.1 Anti-Emulation-Detection

Virtual environment-based dynamic analysis techniques described before can provide effective detection against potential malware. However, smart malware developers can evade dynamic analysis in the virtual environment by detecting the emulator. Dynamic analysis tools must provide anti-emulation-detection defense mechanisms to out-smart the emulation-detection techniques employed by the malware. Malware can detect emulated devices by employing one or more emulation-detection techniques available in emulation-detection library EmuDetLib [100] (see Section 3.1.1).

In an ideal dynamic analysis framework, malware should not be able to detect the underlying emulated environment. To understand the defense measures opted by well known existing dynamic analysis tools along with vanilla Android emulator (referred to as emulator), we have evaluated them against the EmuDetLib. Apart from the EmuDetLib, we have randomly selected 1000 malware samples from the D2:AndroZoo-2019 dataset (see Section 3.2.2) and also downloaded motion sensor's malware disclosed by the Trend Micro [76] (referred to as RealMal) to evaluate the existing frameworks.

Further, to evaluate GPS information based detection and distributed detection, we need a different environment. For GPS, we require a fake GPS location generation App inside an emulated device. For distributed detection, we need a server where the emulation-detection method is deployed and requires multiple instances of the same sandbox running at the

same time. We utilize the command and control server of the real malware Dendroid [101] by employing the emulation-detection Algorithm 2. Table 4.1 shows the evaluation result of the emulation-detection of candidate sandbox against all the detection methods available in EmuDetLib. As shown in Table 4.1, in distributed detection, when the server is configured with the emulation-detection method, none of the frameworks can hide their emulated environment. Similarly, in GPS-based detection, only with a fake app installed emulated-platform can bypass the detection mechanism. In other cases of GPS, the sandbox is flagged as an emulated platform by the detection library.

TABLE 4.1: Defense mechanisms provided by existing dynamic analysis tools against different types of emulation-detection methods EmuDetLib.

| **Detection Type** | **Emulator** | **Droidbox** | **CuckooDroid** | **MobSF** | **DroidScope** |
|---|---|---|---|---|---|
| UDI Basic | × | ✓ | ✓ | ✓ | ✓ |
| UDI Smart | × | × | × | × | × |
| Sensors | × | × | × | × | × |
| Device State | × | × | × | × | × |
| GPS | × | × | × | × | × |
| Distributed | × | × | × | × | × |

*Note:* ✓ represents successful in bypassing the emulation-detection attack by underline emulated, whereas × represents failure in bypassing the emulation-detection attack. UDI represents Unique Device Information. We use this notation in the rest of the tables.

TABLE 4.2: Evaluation of existing framework against real malware sample.

| **Detection Type** | **#Sample** | **Emulator** | **Droidbox** | **CuckooDroid** | **MobSF** | **DroidScope** |
|---|---|---|---|---|---|---|
| No emulation-detection | 284 | ✓ | ✓ | ✓ | ✓ | ✓ |
| UDI | 137 | × | ✓ | ✓ | ✓ | ✓ |
| File Info / SysProp | 303 | × | ✓ | ✓ | ✓ | ✓ |
| Device State | 19 | × | × | × | × | × |
| Sensors | 3 | × | × | × | × | × |
| Mix | 257 | × | ✓ | ✓ | ✓ | ✓ |

*Note:* ✓ represents successful in bypassing the emulation-detection attack by underline emulated, whereas × represents failure in bypassing the emulation-detection attack. UDI represents Unique Device Information. SysProp represents the system properties. The Mix represents the malware sample that uses more than one detection method from Unique Device Information, File Info and System Properties.

Similarly, on executing samples of RealMal (see Table 4.2), Android SDK emulator cannot hide its emulated environment against malware samples with emulation-detection capability. Simultaneously, other sandboxes get detected by the malware samples under the category of device state and sensors. To reason about such behavior, we have investigated the malware sample (RealMal) under sensor category. Listing 4.1 shows the code snippet from BatterySaverMobi (a real malware), which uses accelerometer (line 5) reading to observe motion on a device. If any motion takes place, then it executes the malicious

code (line 15). Hence, Such malware can bypass the dynamic analysis job performed on existing sandboxes.

```
1  public void onSensorChanged(SensorEvent paramSensorEvent) {
2    this.k.registerListener(this, this.l, 3);
3    Sensor sensor = paramSensorEvent.sensor;
4    this.k.registerListener(this, sensor, 3);
5    if (sensor.getType() == 1) {
6      float[] arrayOfFloat = paramSensorEvent.values;
7      float f1 = arrayOfFloat[0];
8      float f2 = arrayOfFloat[1];
9      float f3 = arrayOfFloat[2];
10     long l = System.currentTimeMillis();
11     if (l - this.m > 100L) {
12       long l1 = this.m;
13       this.m = l;
14       if (Math.abs(f1 + f2 + f3 - this.n - this.o - this.p)
   / (float)(l - l1) * 10000.0F > 600.0F)
15         a();
16       this.n = f1;
17       this.o = f2;
18       this.p = f3;
19     }
20   }
21 }
```

Listing 4.1: Code snippet from motion sensor malware.

**Summary of Anti-Emulation-Detection:** Some key observations regarding the effectiveness of anti-emulation-detection measures of the existing analysis platforms against EmuDetLib are shown below.

*(i)* Existing analysis frameworks are able to bypass the basic emulation-detection techniques based on unique device information. However, they fail to defend when the emulation detection attacks are performed by analyzing the underlying defense mechanism. The main reason being either the data is unrealistic (basic detection) or the data is realistic but static (smart attack).

*(ii)* Every framework fails to defend against the emulation-detection attacks based on fluctuating sensors and GPS data since the data does not represent the realistic behavior of a device.

*(iii)* Similar to the detection methods based on UDI, existing frameworks are also not able to defend against distributed emulation-detection. The observation of similar data for unique device-related information across multiple devices helps in raising the red-flag regarding the underlying emulated environment.

*(iv)* Detection methods based on the device state (e.g. telephony system, battery power) also successfully detects the underlying emulated environment due to the absence of defense mechanisms in the analysis frameworks.

In short, the extended emulation-detection techniques show that the existing publicly accessible dynamic analysis frameworks do not provide foolproof anti-emulation-detection measures. Therefore, there is a need for a robust anti-emulation-detection approach that can hide the underlying platform from smart emulation-detection measures. *Our proposal (STDNeut, Section 4.4 and ARTmon, Section 4.5) bridges this gap by employing intelligent anti-emulation-detection measures in the emulation platform.*

### 4.3.2 System Call Monitoring

Tracking app interaction with the OS through system calls can provide useful insights regarding the app behavior. `Strace` [92] is a well known Linux utility available in all flavor of Linux based operating systems including Android OS. A typical usage of `strace` is to execute an app using `strace` to capture all the system calls used by the app from the beginning. However, this is not the case with the Android OS due to the following reasons.

Unlike application on traditional computer systems where application entry point is fixed (e.g., the main() function), an Android app has multiple entry points. Execution of Android app starts from one of the multiple entry points by sending an intent (a.k.a. the message passing technique in Android) from the package manager. In such a scenario, `strace` is unable to capture the system calls used by an app from its start.

To capture system calls invoked by an app from the beginning of its execution, one can apply `strace` on the Zygote process, which is responsible for setting up the runtime environment for the apps and create new processes. When `strace` is applied on the Zygote process, it not only captures the system calls made by apps of interest, but also the system calls made by other apps not of interest. System-wide system call logging results in a lot of overheads, both in terms of storage and processing power. Moreover, all system calls are not useful for malware detection, so researchers are more interested in profiling a set of system calls for further analysis. Eventhough `strace` is capable of profiling a selected set of system calls, it also profiles the same set of system calls made by other apps. This results in unwanted system call in the log file and requires parsing to extract app specific system calls which can be a lot of overheads and efforts. To show the overheads of `strace` when used in Android platform we have performed the following experiment.

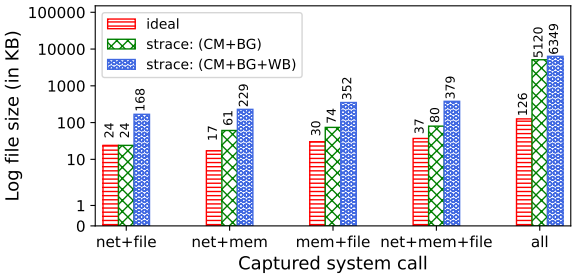



FIGURE 4.3: Storage overhead of `strace` based system call logging w.r.t. ideal (targeted) logging using CaffeineMark (CM) along with background apps (BG) and a web browser (WB). The lower the better.

We executed Java based micro benchmark CaffeineMark [102] and captured system call information using `strace`. We have executed CaffeineMark with two scenarios: *(i)* CaffeineMark with background processes and, *(ii)* CaffeineMark with a browser app along with other background processes. We repeated the experiments to capture combinations of different categories of system calls in the above scenarios and compared the capture log size with the ideal capture log size. The ideal capture log size represents the size of the log file when the system call tracing is performed in a targeted manner (similar to proposed solution). Figure 4.3 compares the ideal log file size with size of the log file generated

by `strace` in different setups. When all system calls are profiled, the ideal log file size is ∼40X and ∼50X smaller than the log generated by `strace` when CaffeineMark is executed along with background processes and with browser app, respectively. Even when a subset of system calls are captured with only background processes, `strace` results in up to 3.5X increased log file size (Figure 4.3) compared to ideal profiling. Therefore, a sophisticated and low-overhead system call profiling utility can improve the efficiency of dynamic analysis in emulated platforms.

Apart from the logging overheads associated with `strace`-based technique, there are some other challenges to analyze logs generated by `strace`. In `strace`, process ID is the only available filter that can be directly applied to parse the log files. However, the process ID cannot be mapped to a specific app from the log file itself and requires support from other utilities like `ps`, `top` etc. Further, when `strace` is used to follow child processes (which is the case with Zygote), there are incomplete log entries because of overlapping system calls from multiple processes.

Furthermore, a malware can easily bypass the `strace` based analysis process by launching `strace` on itself. In this case, given that the `strace` based dynamic analysis system is already tracing the malware, the `strace` launched by the malware will fail, which exposes the dynamic analysis process.

To address these issues related to system call profiling, as part of InviSeal, we propose SysCallMon (Section 4.6), a Linux kernel extension (kernel module), for low-overhead targeted system call profiling.

### 4.3.3 Memory Forensics

Memory forensics plays an essential role in finding evidence from the volatile memory in investigating Cyber Crime. Nowadays, malware authors use advanced techniques such as code packing, dynamic code loading, etc., to bypass static analysis. In the dynamic code loading method, malware loads the code at runtime, which we can obtain from the

volatile memory. Similarly, a packed malware first requires the unpacking of packed binary to perform a malicious action. This unpacked code and decryption keys are available in the memory during the execution of malware. The memory forensics approach can be useful in such a scenario to retrieve the evidence (unpacked code, dynamically loaded code) from the volatile memory, and further analysis can be performed on such code to capture the behavior of malware. Keeping this in mind, the researcher has started working on the memory forensics-based malware analysis [103]. In [104], authors have presented the effectiveness of memory forensics in the malware analysis process. Hence, a dynamic analysis tool must support an on-demand functionality to capture volatile memory for further analysis.

## 4.4 STDNeut: Design & Implementation

First, in this section, we talk about how to make realistic sensor data and the challenges associated with it. Then, we give an overview of STDNeut, a detailed anti-emulation-detection system, and elaborate the design of its various components. STDNeut aims to neutralize emulation-detection using different sensors, telephony system, and device state data.

### 4.4.1 Realistic Sensor Data Generation

A smartphone contains multiple sensors (e.g., accelerometer, GPS, and others) or interacts with an external entity like BTS. A malware can use these sensors to detect an emulated environment. To nullify the effect of sensors based emulation-detection, we have identified three main challenges as follows:

*(i)* Existing sensors value should fluctuate with respect to time.

*(ii)* Detection of emulation environment through sensor correlation.

*(iii)* Model should be flexible to incorporate new sensors and sensor relations.

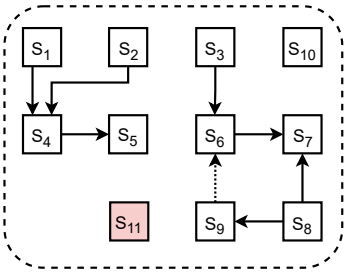



FIGURE 4.4: An example of sensor's dependency graph. Sensor $S_{11}$ in shaded box represents a new sensor introduced in the system.

To better understand these challenges, let us take a directed graph shown in Figure 4.4 that represents eleven sensors ($S_1$ to $S_{11}$), and influence of one sensor on others in terms of driving the sensor's values. An arrow from sensor $S_i$ to sensor $S_j$ denotes that the value of sensor $S_j$ depends on the value of sensor $S_i$. If we see an update in the value of sensor $S_i$, then sensor $S_j$'s value should also be seeking an update according to $S_i$'s value. As shown in Figure 4.4, some sensors do not depend on other sensors (sensor with zero in-degree); we name them as independent sensors, whereas sensors with in-degree $\geq 1$ are called dependent sensors because the value of these sensors depends on the value of others.

Challenge *(i)* is easy to understand, which states that the value of the sensor should fluctuate concerning time. For example, let us consider sensor $S_{10}$ (assuming as a light sensor) in Figure 4.4, the value of this sensor should be updated according to the operating environment lighting condition. Similarly, other sensor's value should also be updated w.r.t. time or working environment condition.

To understand challenge *(ii)*, consider two sensors $S_4$ (assuming as GPS) and $S_5$ (assuming as BTS). As shown in Figure 4.4, sensor $S_5$'s value depends on the value of sensor $S_4$. This dependency is based on the distance between the values of $S_4$ and $S_5$, which cannot be more than $x$ meters. This $x$ may vary depending on the area density (population and obstacles) of the BTS. Further, to be more clear about challenge *(ii)*, let us include two more sensors $S_1$ (as time) and $S_2$ (as an accelerometer). The value of sensor $S_4$ depends on both the sensors, i.e., $S_1$ and $S_2$. If we consider time and GPS, then there is a correlation between the current GPS location and the previous location w.r.t. time elapsed. For example,

if the current GPS location is Washington DC, a person cannot reach New York in five minutes. Similarly, when considering accelerometer and GPS, then the measurement of the distance travel through accelerometer should match with the distance between two consecutive GPS locations. Hence, a sensor-based anti-emulation-detection system should be compliance to all these scenarios so that the use of sensor's value in an innovative way (as described above) cannot reveal the identity of the underlying system.

Challenge *(iii)* is related to the introduction of a new sensor into the system. If a new sensor is included in the system, either it is an independent or dependent sensor (sensor $S_{11}$ as shown in Figure 4.4), the system should be flexible to reprogram so that new sensors can also be adopted for providing anti-emulation-detection capability.

To emulate realistic values for sensors, one should consider all the scenarios, as discussed above. Hence, a fine-grained method is needed to emulate sensors reading while maintaining the dependencies between them along with the re-programmable capability to adopt new sensors in the system.

To address all the challenges as mentioned above, we present Algorithm 3, which takes two lists. One list holds the available sensor object ($sensors_{obj}$) and the other is related to the dependency between sensors ($dependSens_{obj}$). A sensor's object comprises of sensor's identity (like accelerometer, GPS), a default handle and the initial value. The default handle is useful when a sensor does not depend on others (independent sensors), and the initial value is used to initialize the sensor. On the other hand, a dependency object comprises the identity of two sensors $S_i$ and $S_j$, and a dependency function $F_{ij}$, which represents the dependency between $S_i$ and $S_j$. These two lists have to be provided by a user, and Algorithm 3 generates an ordered list of sensors handle ($sensors_{hndl}$), which can be executed at the analysis time to emulate the sensor's value while preserving the relationship between them.

In Algorithm 3, $Unprocessed_{chld}$ denotes a queue of sensors whose immediate child needs processing w.r.t. its handle to emulating the sensor value, whereas $Processed_{chld}$ holds the

**Algorithm 3:** Generate Handle for Sensors

**Input :** $sensors_{obj}, dependSens_{obj}$ `// List of sensors and dependencies objects`
**Output:** $sensors_{hndl}$ `// Ordered list of handles to generate realistic sensors values`

```
1  sensors_hndl ← φ
2  Unprocessed_chld ← φ // Sensors queue whose child is not processed
3  Processed_chld ← φ // List of sensors whose child is already processed
4  Dependecy_graph ← generate_graph(dependency_obj, sensors_obj)
5  Independent_sensors ← getZeroInDegreeNodes(Dependecy_graph)
6  foreach S in Independent_sensors do
7  |  S_hndl ← default_hndl(sensors_obj, S)
8  |  append(sensors_hndl, (S, S_hndl))
9  |  append(Unprocessed_chld, S)
10 while ¬(empty(Unprocessed_chld)) do
11 |  S ← dequeue(Unprocessed_chld)
12 |  childs ← getChilds(Dependecy_graph, S)
13 |  foreach C in childs do
14 |  |  dep_func ← getDep_func(dependency_obj, (S, C))
15 |  |  C_hndl ← generate_hndl(sensors_obj, C, dep_func)
16 |  |  if C not in sensors_hndl then
17 |  |  |  append(sensors_hndl, (C, C_hndl))
18 |  |  else if C is in Processed_chld then// Handling cyclic dependency
19 |  |  |  dep_func ← getDep_func(dependency_obj, (S̄, C))
20 |  |  |  C_hndl ← generate_hndl(sensors_obj, C, dep_func)
21 |  |  |  update_hndl(sensors_hndl, (C, C_hndl))
22 |  |  else
23 |  |  |  update_hndl(sensors_hndl, (C, C_hndl))
24 |  |  if C not in Unprocessed_chld and C not in Processed_chld then
25 |  |  |  append(Unprocessed_chld, C)
26 |  append(Processed_chld, S)
27 return sensors_hndl
```

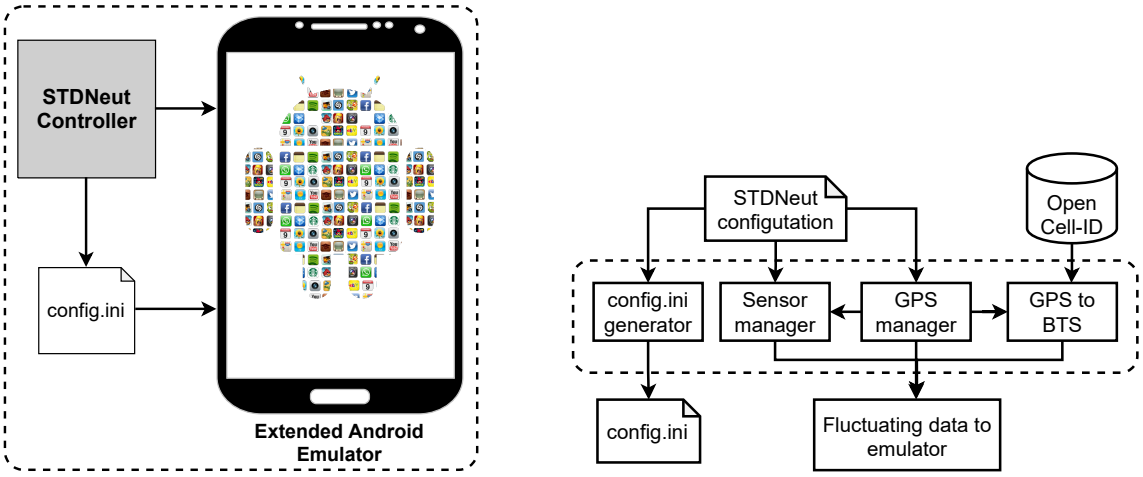
list of sensors whose child has already been processed. Apart from storing processed sensors, the algorithm utilizes this list to break any cyclic dependency (see dependency among sensors $S_6$ to $S_9$ in Figure 4.4), which is a rare case for sensors. As shown in line 4, the algorithm generates a dependency graph among sensors by using the list of $dependSens_{obj}$ and $sensors_{obj}$. Line 5 gets the list of independent sensors from the dependency graph from where actual learning of sensor handle starts. From lines 6 to 9, the algorithm obtains a handle for each independent sensor, which is equivalent to the default handle in sensor object. The default handle is used to generate the value for a sensor, which does not depend on other sensors. Apart from the sensor handle, independent sensors are then

appended in the $Unprocessed_{chld}$ queue, because the children of these sensors may require a handle.

From lines 10 to 26, the algorithm generates the handles for the dependent sensors. The algorithm terminates when the $Unprocessed_{chld}$ queue does not contain any sensor for processing. Line 14 gets the dependency function between parent sensor $S$ and the child sensor $C$ by using the $dependSens_{obj}$ and a handle gets generated at line 15. At line 16, it checks if the sensor is not in the list of $sensors_{hndl}$, algorithm directly adds this handle into $sensors_{hndl}$. In other cases, it updates the already learned handle based on the current dependency and the dependency learned earlier. For updating an already learned handle, there can be two possibilities, one is related to cycle (see cyclic dependency in Figure 4.4 among sensors $S_6$ to $S_9$) and the other is when a sensor depends on more than one sensor (See sensor $S_4$ in Figure 4.4). A cyclic dependency is resolved at line 18 in Algorithm 3, where a new dependency function is calculated between parent $S$ and child $C$. To obtain the new dependency function, we utilize the last value of $S$ (referred to as $\bar{S}$ in line 19) to update the handle of $C$. At last, when all the children of a sensor $S$ are processed, $S$ is added to the $Processed_{chld}$ at line 26. Finally, the algorithm returns an ordered list of $sensors_{hndl}$, which is then used to emulate the sensor's value at run-time. This algorithm handles the challenge *(i)* and *(ii)*. For challenge *(iii)*, if the user updates the list of sensor objects and dependency objects, then it re-generates the sensor handles for all the sensors, including the new sensors.

### 4.4.2 STDNeut Overview

STDNeut provides robust support for anti-emulation-detection that can be used to design an efficient framework for malware analysis. Figure 4.5 shows the architecture of STDNeut along with the design of its controller. As shown in Figure 4.5(a), there are two main subsystems of the STDNeut: *(i)* Extended Android Emulator and *(ii)* STDNeut Controller (see Figure 4.5(b)). We describe the design of the subsystems in this section.




(a) STDNeut design overview.

(b) Design of STDNeut controller.

FIGURE 4.5: Architecture of STDNeut, an anti-emulation-detection system along with the STDNeut controller.

**Extended Android emulator:** It is responsible for spoofing the information related to sensors, telephony systems, and device data. The STDNeut controller and `config.ini` file govern this spoofing information to the Android emulator. Most of the device-specific information, like IMEI, remains constant during the execution time, while the values for sensors and telephony signal fluctuates over the time. During the boot time, the Android emulator reads `config.ini` file and configures a virtual device with device-specific information that is unique to it, while the STDNeut controller handles the fluctuating values at run-time.

**STDNeut controller:** It is responsible for launching an app inside the emulator and feeding essential information for anti-emulation-detection. For example, the controller generates a `config.ini` file that is being used by the Android emulator to configure a virtual device with unique values. The controller also manages the hardware/environment generated events that alter the state of an Android device such as available sensors, telephony signal, and many more. This is achieved by frequently feeding-in realistic sensor data while maintaining the correlation with other sensors (as described in Section 4.4.1 by utilizing Algorithm 3) and other hardware related events into the emulator. To feed the sensor data and hardware-related events, the controller uses the emulator console

APIs [105]. Other than the core features mentioned above, the controller also enables and configures other functionalities which simulate incoming calls/SMSes, manipulates signal strength, and many more. We discuss the extension made to Android emulator in the next section.

### 4.4.3 Extensions to the Android Emulator

A smartphone contains multiple sources of information that are either unique to a device and does not change during its life or information may get changed over time due to the operating environment that alters the state of it. Mostly, a device gets a unique identity from the telephony system that includes IMEI, IMSI, phone number, and many more. To interact with the telephony system, AT commands [106] are being utilized. To provide a unique identity to a virtual device, we intercept the AT command request at the emulator layer for spoofing the response. For example, a smartphone makes "AT+CGSN" and "AT+CIMI" commands to query IMEI and IMSI numbers, respectively. This spoofed information is fed to the AT command by concerning the `config.ini` file. Similarly, other values are also being fed in response to the AT commands that remain constant but unique to a device. Apart from the `config.ini` file, these values can also be supplied to a virtual device using command line arguments. For the hardware/environment events that alter the device state, we use the emulator console to supply realistic data periodically. The Android emulator itself provides most of the hardware like sensors, GPS, signal strength, and others; the data for them can be fed by using emulator console at run-time. Android emulator does not provide any interface to change the BTS information with whom a device is currently associated. To provide a realistic GPS location, the information about the BTS associated with the device should collaborate. In this observation, we have added the BTS information alteration interface through the emulator console, and the STDNeut controller is supplying the realistic BTS identity. We discuss the detailed design of STDNeut controller in the next section.

### 4.4.4 STDNeut Controller

The primary responsibility of STDNeut controller is to generate `config.ini` file and feed-in the realistic values for the fluctuating sensors and other hardware events. As shown in Figure 4.5(b), the STDNeut contains four core components: *(i)* config.ini generator, *(ii)* sensors manager, *(iii)* GPS manager, and *(iv)* GPS to BTS.

**config.ini generator**: It generates the `config.ini` file to spoof device-specific unique information.

**Sensor manager**: It manages the device sensors by feeding-in realistic data periodically. To generate the value of sensors, it uses the same handles which are obtained through the Algorithm 3. The sensor manager manages all the sensors and other hardware events except the GPS. However, it gathers the next GPS coordinate to be projected by GPS manager so that the sensors on which GPS depends, can generate appropriate values.

**GPS manager:** The main reason behind the separate manager for the GPS is the correlation between the current GPS location and the previous location. For example, if the current GPS location is Washington DC, then it is impossible that a person can reach New York in five minutes. Hence, a random GPS location alerts an app about the emulated environment. So a precise method is required to feed GPS location to an emulated environment, and GPS manager provides the same. The GPS manager reads the source and destination geo-location along with the travel time from the STDNeut configuration file and generates a route by using a path patching algorithm, as shown in Algorithm 4. This algorithm takes source and destination geo-locations along with the number of steps required to move from source to destination, and returns the route trajectory.

**GPS to BTS:** A realistic GPS location alone is not strong enough to hide an emulated environment. It must be assisted by the BTS location that correlates with the current GPS location. This correlation is based on the maximum distance between the BTS and GPS locations that may vary from 1 km to 3 km depending on the area density in terms of population and obstacles. There are several commercial and public services that provides

**Algorithm 4:** Path patching for GPS trajectory

**Input :** $Lat_{src}, Long_{src}, Lat_{dst}, Long_{dst}, nSteps$
**Output:** $trajectory$

$trajectory \leftarrow \phi$
$LatStep_{max} \leftarrow |Lat_{src} - Lat_{dst}| \ / \ nSteps \times 2$
$LongStep_{max} \leftarrow |Long_{src} - Long_{dst}| \ / \ nSteps \times 2$
$Direct_{lat} \leftarrow +1 \ if \ Lat_{dst} > Lat_{src} \quad else \ -1$ // direction
$Direct_{long} \leftarrow +1 if \ Long_{dst} > Long_{src} \quad else \ -1$
$(lat, long) \leftarrow (Lat_{src}, \ Long_{src})$
$append(trajectory, (lat, long))$
**foreach** $i \ in \ range(0, \ nSteps)$ **do**
    $lat \leftarrow lat + rnd.uniform(0, \ LatStep_{max}) \times Direct_{lat}$
    $long \leftarrow long + rnd.uniform(0, \ LongStep_{max}) \times Direct_{long}$
    $append(trajectory, \ (lat, \ long))$
**return** $trajectory$

the GPS location by using a BTS ID. Still, no one provides the reverse mapping of it, i.e., providing a BTS ID based on GPS location and the SIM operator that is closer to the current GPS location. GPS to BTS module bridges this gap with the help of the OpenCellID database [107]. The OpenCellID database contains information for the already installed BTS, worldwide, which is publicly available for research purposes. As this database stores BTS information worldwide, an efficient search mechanism is needed to retrieve BTS ID that operates based on the current GPS location and SIM operator. With this observation, we first filter the database based on the mobile country code (MCC), followed by the mobile network code (MNC). MCC and MNC reduce the search space to a specific operation within a country. Now we only need location area code and cell-ID to get the desired BTS ID, and that is retrieved by calculating the distance with stored BTS location in the database and the current GPS location, which is compared against the maximum distance allowed. We have used `haversine` [108] to measure the distance between the BTS location and the current GPS location. The main reason for separate module for GPS to BTS correlation is because it requires to interact with external database for retrieving the BTS ID according to the GPS location. In next section, we discuss the design of ARTmon that is designed on the top of STDNeut with extended anti-emulation-detection capabilities that is not supported by the STDNeut.

## 4.5 ARTmon: Monitoring Framework APIs

ARTmon design is based on the `Droidmon` that is developed using the `Xposed` framework. At a high level, ARTmon can be considered to be an enhanced `Droidmon` with a lot of configurability and inbuilt anti-emulation-detection techniques related to the files and system properties. Figure 4.6 illustrates the design of ARTmon using the `Xposed` framework. ARTmon modifies the hooking engine and the hooks config file used by `Droidmon` (Section 4.2.1.1) to incorporate anti-emulation-detection measures. In ARTmon, hooking engine and associated config file will not only provide the information to instrument framework level APIs, but also modifies the return value of an API call as per the configuration. This is useful to provide non-detectable values when system properties and files information is queried from the apps. To provide anti-emulation-detection measures related to system properties and files, ARTmon provides configuration files through which the controller can dictate the values provided to serve app queries. For example, system property manager configuration can contain settings to serve run-time information queries (e.g., whether the platform is a debugger) from the app. Similarly, the file manager can be configured to conceal file related information when queried through APIs like `Exists` and `Open`.

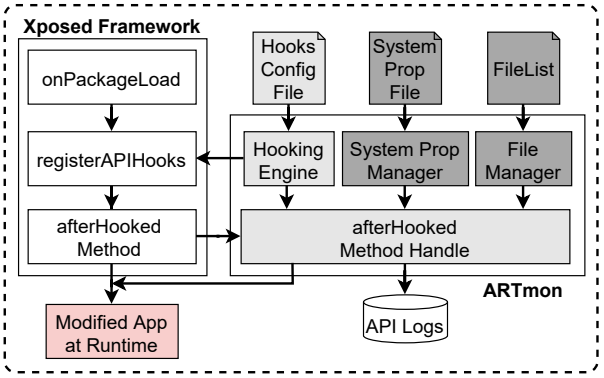



FIGURE 4.6: ARTmon design using Xposed Framework.

A user can modify the three configuration files mentioned above to control the app profiling behavior and enable different anti-emulation-detection measures. At runtime, the afterHooked Method Handle, invoked from the modified apps, consults with all managers:

system prop manager, file manager and hooking engine: to substitute the app behavior as specified in the respective configuration files. For some APIs (e.g. open), beforeHooked method is also used.

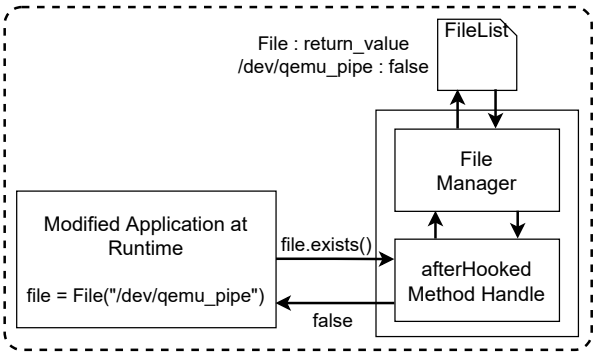



FIGURE 4.7: Working of anti-emulation-detection using ARTmon

Figure 4.7 shows a working example of the anti-emulation-detection techniques supported by ARTmon. As shown in Figure 4.7, when a modified app (using `Xposed`) requests to check the existence of `/dev/qemu_pipe` file (with the intention of detecting emulation), the afterHooked method handle consults the file manager to check the configuration settings. The file manager checks the file name by looking up the configuration file for the available measures related to anti-emulation-detection through file operations. If the requested file is present in the FilesList config file, it provides the configured return value to the file manager (in our case: `false`) which is then communicated back to the afterHooked method handle. According to the response received from the file manager, afterHooked method handle substitutes the behavior of the `FileExists` method. Similarly, ARTmon provides defense mechanisms against the emulation-detection attacks through different app methods using user provided configuration files.

## 4.6 SysCallMon: System Call Monitor

SysCallMon is a configurable kernel module designed to monitor and profile the system calls used by apps in an efficient manner. As shown in Figure 4.8, the SysCallMon controller provides a list of system calls to be profiled for different apps to the SysCallMon

through ADB. One of the design challenges is to filter system calls of interest from profiled apps to reduce overheads mentioned in Section 4.3.2. Process ID (PID) comes as the first choice for this purpose, but fails in this case as process ID is allocated on a `fork()` system call from the parent process. By implication, the complete process tree hierarchy is required to be profiled resulting in unnecessary overheads.

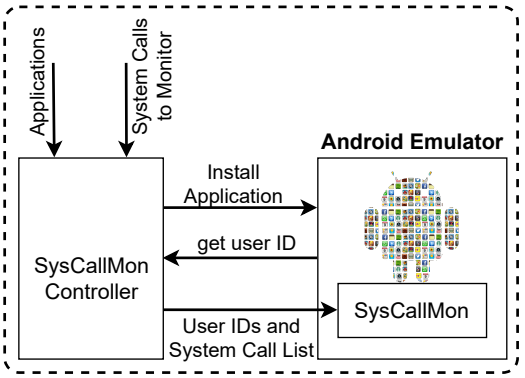



FIGURE 4.8: High-level view of SysCallMon module.

In Android systems, apps can be identified easily through user IDs as different apps are associated with different user IDs. Each app is assigned a unique user ID at the time of installation by the Android system. The SysCallMon controller communicates the user ID to the SysCallMon for targeted system call profiling. Next we discuss implementation aspects including the subtle challenges to design SysCallMon.

### 4.6.1 Implementation

As discussed earlier, the SysCallMon is a kernel module designed to profile the system calls invoked by apps in a configurable manner. Figure 4.9 shows, SysCallMon takes the list of user IDs corresponding to the apps of interest along with the list of system calls to be monitored as input at the time of module initialization. To filter apps under observation, SysCallMon initializes a list containing the user IDs of monitored apps during initialization. Note that, the controller leverages the one-to-one correspondence between apps and the user IDs, and configures the SysCallMon depending on the end-user requirements. Further, during the module initialization, SysCallMon modifies the system call handler table of the

Linux kernel to insert modified system call handlers (used as profiling hooks) for configured system calls and stores the original system call handler functions in a mapping table.

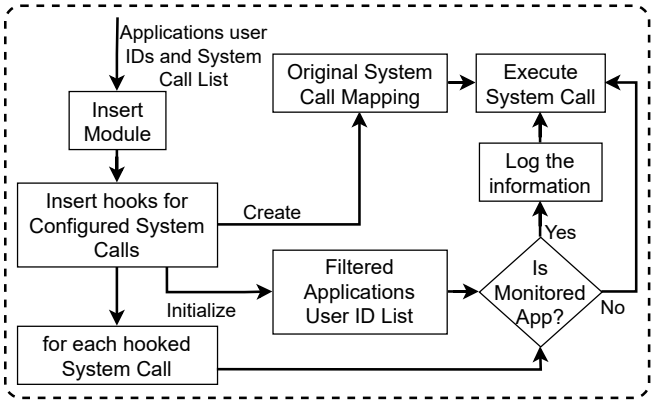



FIGURE 4.9: SysCallMon module work-flow.

Whenever any app invokes a system call, which is monitored (by modifying the system call handler entry), the modified handler in SysCallMon module is invoked. If the user ID of the current context does not match any of the configured user IDs, the original system call handler is invoked. Otherwise, the system call details including the arguments are logged before executing the original handler. Note that, any system call which is not profiled, SysCallMon does not introduce any change to its original behavior in an unmodified system. Further, for monitored system calls originating from apps that are not of interest, very minimal check (UID check) is required before passing them on in the normal execution path.

To use SysCallMon module on an Android emulator platform, there are several challenges as mentioned below.

**Loadable kernel module support:** Android emulator kernel provided by the Android SDK does not support loadable kernel modules. Therefore, we cannot load our SysCallMon module to profile system calls. To overcome this issue, we have downloaded the Android emulator kernel (Goldfish/Ranchu) source code and compiled it after enabling the loadable kernel module support.

**Visibility of system call table:** In the recent Linux kernel releases, system call table address is not visible from the user space because of security reasons. Earlier, one could read the address of system call table from the `System.map` file which is located in the boot partition. Android emulator does not have a separate boot partition and we could not get the address of the system call table. In our custom built kernel, we extract the system call table address by reading the `System.map` file generated during the build process.

**System call table page is write protected:** In recent Linux kernel, the system call table is mapped as write-protected even for the kernel mode access to avoid accidental overwrite of the original system call handlers. Therefore, write protection does not allow our module to modify system call handlers with our hooks for monitoring system calls. To overcome with this problem, we temporarily disable write protection and reinstate it after modifying the system call handlers for configured system calls. Note that, the same process is repeated to reinstate the original system call handlers during module unload.

## 4.7 InviSeal: Building the System

So far, we have designed the anti-emulation-detection techniques using the STDNeut (Section 4.4) and ARTmon (Section 4.5) to bypass majority of emulation-detection attacks, including sensors, files, and system properties. Further, we have designed an OS-level system call interposition technique (SysCallMon, Section 4.6) to profile the system call activities while incurring low profiling overhead. In this section, first we discuss the requirement of an integrated solution to analyse malware. After that, we provide an overview of the InviSeal, a comprehensive Android security audit framework designed based on STDNeut, ARTmon and SysCallMon (Section 4.6).

### 4.7.1 Why An Integrated Solution is Required?

Malware analysts can perform multiple types of analysis ranging from cross-layer profiling to memory forensics to identify malicious behavior of an app while hiding the underlying

emulated environment. However, none of the solutions which we have designed earlier (i.e., STDNeut, ARTMon, and SysCallMon) can provide all functionality in one place. For example, STDNeut alone cannot provide framework-level profiling information. It only protects the emulated environment from being detected as an analysis platform by using sensors, telephony systems, or device state information. Similarly, ARTmon alone cannot be used for profiling such malware that includes malicious behavior inside the native code. Moreover, SysCallMon alone neither protects emulated platforms from being detected as an analysis environment nor profile framework-level APIs. Furthermore, none of these solutions provides the ability to perform malware analysis based on memory forensic techniques. Hence, an integrated solution is required that can be used in many malware analysis scenarios without being detected as an emulated platform.

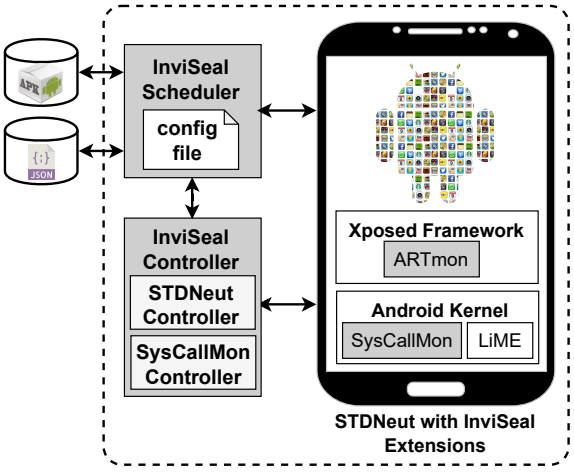



FIGURE 4.10: Architecture of InviSeal, a stealthy dynamic analysis framework for Android systems.

### 4.7.2 An Overview

InviSeal provides an efficient and easy to use end-to-end Android analysis platform with comprehensive support for anti-emulation-detection. Figure 4.10 shows, there are three main subsystems of InviSeal: *(i)* STDNeut with InviSeal extensions, *(ii)* InviSeal Controller, and *(iii)* InviSeal Scheduler. This section describe the design of the subsystems.

**STDNeut with InviSeal extensions** is responsible for providing cross-layer profiling capability including the framework layer and the system call layer. ARTmon is one of the essential modules in the InviSeal, which we have designed using the `Xposed` framework to profile the framework layer APIs. ARTmon along with the STDNeut provides comprehensive anti-emulation-detection functionality to bypass majority of emulation-detection tricks employed by the malware developers. We introduce Linux kernel modifications through a kernel module (referred to as SysCallMon) to profile the system calls in a targeted manner. We also use LiME (Linux Memory Extractor) [109] to capture the entire memory of an Android device, which can be further analyzed to extract useful information such as dynamically loaded code or unpacked code from a packed malware.

**InviSeal Controller's** responsibilities include launching apps inside the Android virtual device by providing necessary information for app execution and profiling. For example, InviSeal controller provides the hooks configuration file required by ARTmon, system call profiling filters like app user IDs and system call numbers to SysCallMon with the help of SysCallMon controller. To control the profiling characteristics the InviSeal controller makes use of the Android Debug Bridge (ADB) [110]. Apart from launching apps, InviSeal controller also manages the sensors, telephony systems and device state information in Android virtual device with the help of the STDNeut controller. Note that, the ARTmon provides most of the anti-emulation-detection mechanisms except for sensor, telephony system and device state information related emulation-detection that is handled by the STDNeut (see Section 4.4). This is achieved by frequently feeding-in realistic sensor data from the STDNeut controller to the emulator. To feed the sensor inputs, the STDNeut Controller uses the emulator console [105] APIs. Hence, ARTmon along with STDNeut provides comprehensive anti-emulation-detection functionality to bypass majority of emulation-detection attacks. Other than the core features mentioned above, the InviSeal controller also enables and configures other functionalities like screen-shots, screen recording, network capture, memory dump etc.

**InviSeal Scheduler** provides a single-point dynamic profiling and analysis support to the

end-user. The end-users submit different apps for security analysis (referred to as analysis jobs) through external APIs. The main responsibility of the scheduler is to schedule analysis jobs on available computing resources by downloading the apps and provisioning InviSeal controller as per the configuration mentioned in the 'config' file. This configuration file 'config' is the master 'config' file for InviSeal, which is used to configure STDNeut and SysCallMon. The computing resources are managed in the units of Android Virtual Device (AVD), provisioned before an analysis job is scheduled and released when the job finishes. It also captures the profiling information from the InviSeal controller and stores them for further analysis.

## 4.8 Evaluation

We use Android Open Source project (AOSP-7.1) to evaluate InviSeal. To overcome the compatibility of apps on x86 emulators having native code (a piece of code written in C/C++), we use ARM address translation library (Houdini) [111]. We use custom built Android goldfish kernel (v3.10) to integrate and test SysCallMon. For the experiments, Android Virtual Device (AVD) instances were configured with two CPU cores, 1.5 GB of RAM, 2GB of internal storage and a 512 MB of SD card along with all the sensors

### 4.8.1 Performance Overhead Analysis

To empirically analyze performance overhead of InviSeal, we ran CaffeineMark-3.0 [102] with and without ARTmon where `baseline` refers to original emulator without any modifications and `ARTmon` refers to emulator with the proposed framework-level API profiling (Figure 4.11). Further, we enabled features like `strace`-based profiling and SysCallMon in both the baseline system and with ARTmon to study the additional overheads due to system call monitoring. Note that, `baseline+Strace` refers to a scenario without ARTmon-based profiling but with strace-based system call profiling. The other settings are to be interpreted in a similar manner. We repeated each experiment on a single AVD

for ten times and record the average score reported by CaffeineMark, where higher score represents lower overheads.

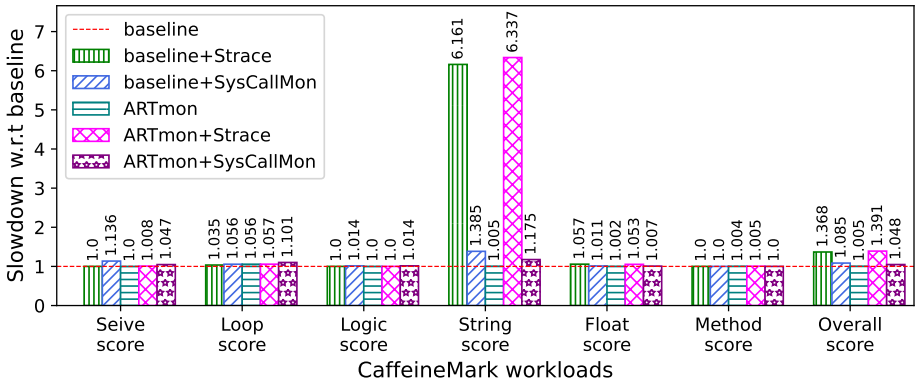


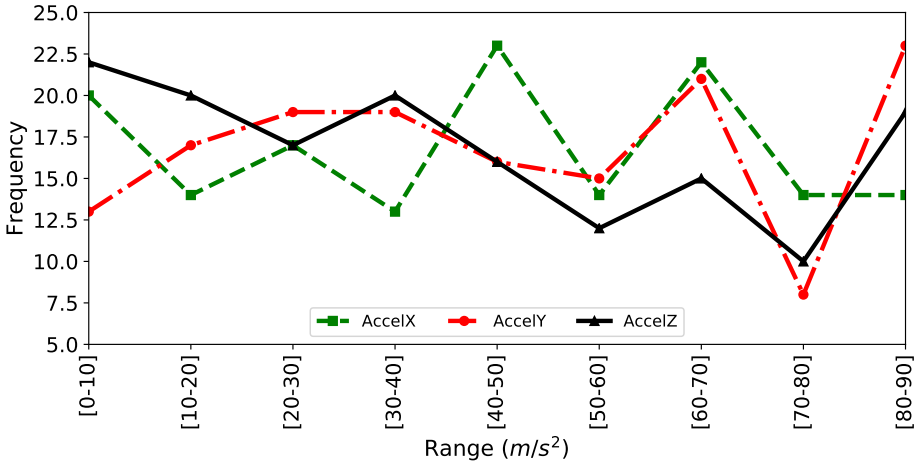

FIGURE 4.11: System slowdown w.r.t baseline (original Android emulator) due to the profiling overheads in different settings using CaffeineMark-3.0 benchmark score. The lower the better.

Figure 4.11 shows that InviSeal with ARTmon and SysCallMon incurs 1.04X slowdown compared to the baseline system. Only ARTmon without any system call profiling results in similar slowdowns compared to the baseline system. We believe that, `Droidmon` will also result in similar overheads as that of ARTmon as the underlying `Xposed` framework remains the same. When `strace`-based system call profiling is enabled, the additional overhead due to strace is 1.36X and 1.38X with baseline and ARTmon, respectively. SysCallMon is ~1.26X faster than the `strace` in both the cases. This shows that, InviSeal provides cross-layer profiling (using SysCallMon) without incurring any significant additional overheads. Moreover, for benchmark workloads issuing high-volumes of system calls, `strace`-based profiling results in significant overheads (e.g., 5.39X overheads in String-score) compared to SysCallMon.

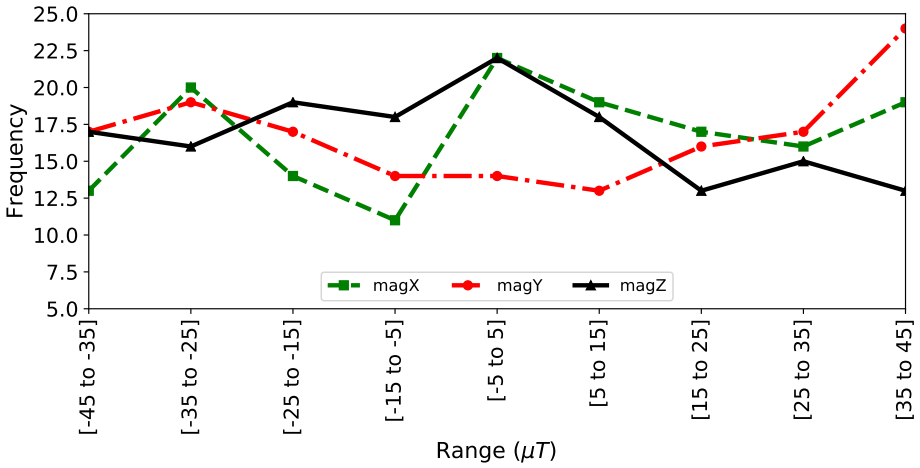
### 4.8.2 Validation of Proposed Anti-Emulation-Detection Measures

**InviSeal vs. EmuDetLib:** We evaluated the effectiveness of the InviSeal against the EmuDetLib (see Section 4.3.1) and found that InviSeal remains undetected against all the attacks performed by EmuDetLib to detect the underlying emulated environment. Similarly, when the malware sample RealMal is used, InviSeal also remains undetected

against all the real malware samples. After evaluating the efficacy of the InviSeal against the EmuDetLib and RealMal, we attempt to understand the reasoning behind this strong defense mechanism by performing various experiments. In the remaining part of this section, we discuss the reasons for the efficacy of InviSeal by analyzing different sensor readings and device information during the experiments.




(a) Distribution of accelerometer readings with anti-emulation measures.




(b) Distribution of magneto-meter readings with anti-emulation measures.

FIGURE 4.12: Effectiveness of InviSeal in neutralizing emulation detection using sensors by providing random reading for accelerometer and magnetometer.

#### 4.8.2.1 Non-detectability through sensors

To evaluate the efficacy of InviSeal against potential malware exploiting sensor readings, we have developed an app to record and store the values of accelerometer, magnetometer, and GPS readings periodically which are shown in Figure 4.12 and Figure 4.13. In this evaluation, we have set two dependencies for sensors, one for time & GPS, and another for GPS & BTS. We make rest of the sensors as independent. The accelerometer reading represents the movement of the device in a three-dimensional space (referred to as AccelX, AccelY, and AccelZ) where the value in each dimension ranges from zero to ninety $(0, 90)$. Figure 4.12(a) shows the distribution of accelerometer readings where the X-axis represents ranges (total of nine ranges) of sensor values and the Y-axis represents the frequency. We have collected the values by executing an experiment for 150 seconds and reading the sensor values every second. The data shows that all the sensor readings are almost equally likely and approximates a random distribution. Therefore, any emulation-detection technique based on accelerometer reading is nullified by our system. For the magnetometer (Figure 4.12(b)), the magnetic field readings on each axis in a three-dimensional system are represented as magX, magY and magZ with a range between $-45$ to $+45$. As shown in the Figure 4.12(b), the distribution is random, thus it does not allow an emulation-detection scheme using magnetometer data to succeed in detecting the underlying emulation platform.

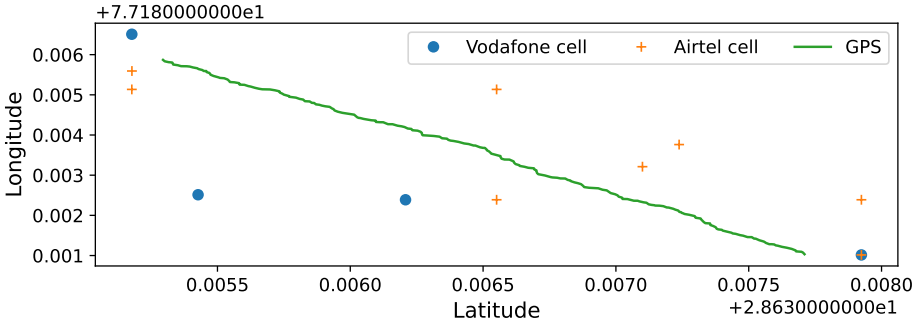



FIGURE 4.13: GPS latitude and longitude reading with anti-emulation measures by feeding-in realistic data along with associated BTS. GPS denotes path trajectory generated using the path patching algorithm.

Another source of emulation-detection is performed by reading GPS data. Unlike accelerometer and magnetometer, GPS data cannot be a random value. Depending on the location of the system, the GPS data should be provided with very slight variations in latitude and longitude. As shown in Figure 4.13, InviSeal anti-emulation-detection measure can provide valid latitude and longitude values along with the associated BTS. In Figure 4.13, GPS denotes the path trajectory generated using path patching algorithm 4 whereas Vodafone cell and Airtel cell denotes the BTS location in the network of Vodafone and Airtel, respectively.

TABLE 4.3: Device information provided by InviSeal with three different AVDs executing the same app.

| **Queried Information** | **Information retrieved** | | |
|---|---|---|---|
| | **AVD1** | **AVD2** | **AVD3** |
| PhoneNumber | 9876543210 | 9856543410 | 9876573213 |
| SimSerialNumber | 89914105611117910720 | 89914145211132510720 | 89914111219018510720 |
| IMSI | 405541385237906 | 405521385237806 | 405511385238906 |
| IMEI | 359470010002931 | 359470010302943 | 359470010002949 |
| SimOperator | 40554 | 40552 | 40551 |

#### 4.8.2.2 Non-detectability through device information

Device information is useful in differentiating between an emulated device and a real smartphone. In emulator platforms, device information such as IMEI, IMSI, phone number etc. are either absent or hold fixed values. To demonstrate the effectiveness of InviSeal's anti-emulation-detection measures, we have used an app called `SIMCardInfo` [112], which extracts the information related to telephony services. We created three instances of this app in three different AVDs and executed all the instances simultaneously for one minute with and without InviSeal. The output of the app queries related to the device information is logged for all instances. We analyzed the log to extract information like IMEI and IMSI. Table 4.3 shows the captured device information with InviSeal. We are not showing the results other than the proposed system as the device readings were the same for all the instances. As shown in Table 4.3, InviSeal is capable of providing a unique device identity in a multi-instance setup. This is particularly useful to avoid detection

when analyzing potential malware running in separate devices designed to operate in a collaborative manner as all malware sees same device identity. However, the values of PhoneNumber, IMEI, and IMSI are generated manually in the experiment which can be configured through the InviSeal's configuration file without any modification in the Qemu.

### 4.8.3 InviSeal Use Cases

We present three use cases of InviSeal to show its effectiveness as a security analysis platform.

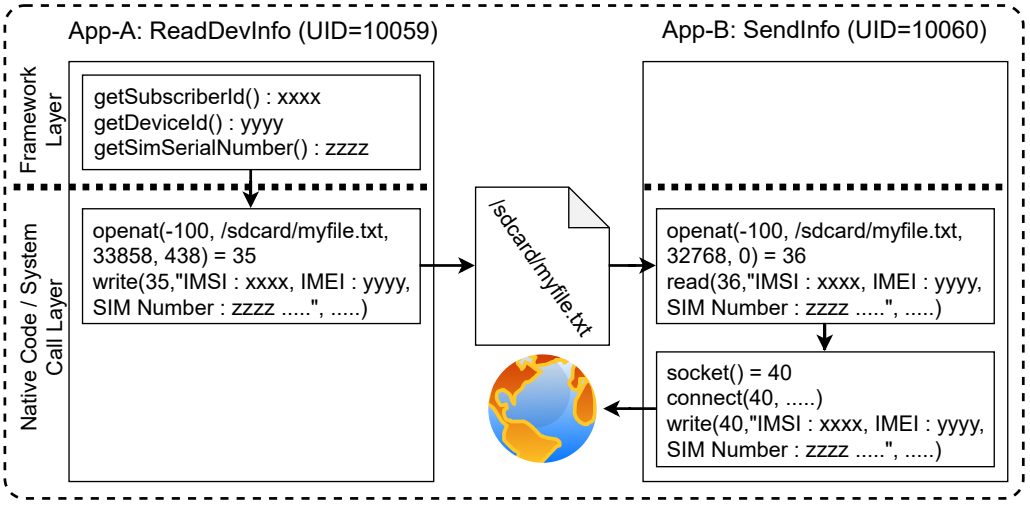



FIGURE 4.14: Detecting collusion attack using cross layer profiling. Dotted lines separate the framework and native layers.

#### 4.8.3.1 Detection of collusion attacks using cross-layer profiling

To show the effectiveness of cross-layer profiling in detection of potential malware, we have developed two apps—*ReadDevInfo* and *SendInfo*. *ReadDevInfo* has permissions to read device information and can write to external storage (in our case the SDCard), whereas *SendInfo* has permissions to access internet and read from external storage (Figure 4.14). During execution, *ReadDevInfo* queried device information (IMEI, IMSI and Sim Serial Number) using framework level APIs and wrote these information to a file `/sdcard/myfile.txt` using native APIs. *SendInfo* read the same file and established a

network connection using socket APIs (part of native library). Note that, *SendInfo* did not use any framework level APIs to read from file or send data across the network.

In this scenario, to detect information leakage through multiple apps, cross-layer profiling is required. Emulator platforms like `Droidmon` provides only framework level activity profiling which is insufficient to successfully tackle this scenario. In this case, `Droidmon` like profiling techniques can follow the device information trail within the framework layer and get constrained because of their inability to extend the monitoring into the lower layers. However, with cross-layer profiling support of InviSeal, we can follow the lead into the captured system calls (by SysCallMon) and flag the malware in a collusive manner. For example, `openat("/sdcards/myfile.txt")` followed by `write()` system call with device information as argument from *ReadDevInfo* is captured by SysCallMon. Additionally, the framework layer also captures the device information API invocation event. When the same file is read from the *SendInfo* app (using `openat` and `read`), the SysCallMon profiles these system calls. A malware analyst becomes suspicious at this point and tracks the other system calls made by *SendInfo* to not only detect the malware, but also the remote entity involved in this activity by monitoring socket system calls (`socket, connect`). Similar information leakage through shared memory and other native IPC mechanisms can also be easily detected by InviSeal.

#### 4.8.3.2 Evading distributed emulation-detection

To show the effectiveness of the anti-emulation-detection measures against emulation-detection using multiple clients along with a central server, we used Dendroid [101], a real Android botnet. We integrated EmuDetLib [100] into the Dendroid malware. We modified the Dendroid control server [101] not to send further instructions to the clients that seem to be running on emulated platforms by observing identical device information like IMEI from multiple clients. Apart from hosting the control server, we also designed a victim site where the malware-infected devices perform a denial of service attack in a distributed manner when instructed from the control server. We created two instances

for each of the CuckooDroid and InviSeal, and then executed Dendroid malware with integrated EmuDetLib. The control server instructs the infected devices to perform an HTTP flood on the victim site mentioned above only if the control server does not detect emulation. In our evaluation, we found that the control server is sending instructions only to the InviSeal system instances and not to the CuckooDroid instances. This was primarily because, the phone number, IMEI, etc. provided to the control server by the CuckooDroid were identical for both the instances, which was not the case with InviSeal. Therefore, we can conclude that the proposed InviSeal system can prevent emulation-detection orchestrated in a distributed setup.

#### 4.8.3.3 Experiments with Memory Dumps

In this experiment, we have selected a set of 50 random malware detected in year 2018. InviSeal is configured to capture the strace and ARTmon logs. Also, two memory dumps were captured for each sample. One dump (initial) was taken before the installation of the sample on AVD, while another one (final) was after executing the sample on the device. We have analysed both the memory dump (i.e., initial and final) by obtaining the difference between them with the help of volatility [113] framework and shown in Table 4.4. During the experiment, 45 samples were able to finish their execution successfully while the remaining 5 samples could not due to the exceeding time limit, hence the final dump has not been collected. Results shown in Table 4.4 represents the average difference in each category. We have not shown such categories whose differences are zero, but they can be obtained from the captured memory dumps, if required.

Table 4.4: Categories wise average difference between initial and final memory dump

| **Category** | **Average number of difference** |
|---|---|
| Process List | 7.289 |
| Process XVIEW | 3.578 |
| PID hash Table | 325.2 |
| NETSTAT | 3.11 |
| Libraries | 1 |
| Hidden DLLs (ldrmodule) | 2340.31 |
| PLT Hooks | 0.022 |
| API Hooks | 0.022 |

Further, memory dump feature can also be used to extract useful evidences/information like app loading dynamic binaries into the memory at execution time i.e., dynamic loaded code or unpacked code of a packed malware. To show the ability to extract such information, we have randomly selected two apps where the information about the dynamically loaded code has been captured by the ARTmon. From the ARTmon log, we found that these apps are loading two files, one is the shared library (.so file), and another one is dex/jar file. To extract these files from the memory dump, we have used volatility tool, and extracted from the final memory dump. The path for these files inside the memory dump and the starting virtual memory address (VMA area) is shown in Table 4.5. An analyst can further perform static analysis to extract more useful information towards the identification of malicious behaviour.

TABLE 4.5: Dynamically loaded file extracted from the memory dump

| App Package Name | File Path | VMA |
|---|---|---|
| com.gwjppa.LZstory | /data/app/com.gwjppa.LZstory-1/lib/arm/libSecShell.so<br>/data/user/0/com.gwjppa.LZstory/.cache/classes.jar | 0xc156000<br>0x946f1000 |
| com.kuman.comic | /data/user/0/com.kuman.comic/.cache/classes.jar<br>/data/app/com.kuman.comic-1/lib/x86/libSecShell.so | 0x97866000<br>x98a62000 |

**Evaluation summary:** In a nutshell, InviSeal can be used to effectively analyse and detect stand-alone and colluding malware in an efficient manner without being detected as an emulated environment.

## 4.9 Related Work

Dynamic analysis techniques for Android app analysis have been drawing the interest of researchers for a long time due to the high use of dynamic code loading and other techniques in the app [32]. Because of these techniques, static analysis fails to capture the behavior of an app.

Dynamic analysis tools based on taint analysis such as taintDroid [64] and taintART [65] are capable of identifying the leakage of sensitive information through the framework level

API. However, such dynamic analysis techniques are not able to capture the information leak that happen through lower level APIs like native code or system calls.

Apart from the taint analysis, framework level API monitoring based dynamic analysis tools have also evolved. Such tools can be designed either by modifying the framework level APIs in Android Open Source Project (AOSP) or by using hooking frameworks like `Xposed` [95]. Hooking framework based approaches are flexible and are easily extendable in comparison to the modification in framework level APIs.

DroidBox [37] is a highly popular and widely used dynamic analysis tool for the Android app, and is based on the framework level API modification in AOSP. Droidbox also provides the functionality of taint analysis by including the taintDroid in it. However, it has been used by most of the researchers. It can only analyse an app that can run till Android version 4.4. On the other hand, the dynamic analysis tools like CuckooDroid [38] and MobSF [39] are based on the `Xposed` (hooking framework) and are capable of hooking the framework level APIs at runtime. These tool use the `Droidmon` [99] module (Section 4.2.1.1) to enlist the APIs used by an app. MobSF provides a virtual machine for Android x86 version 4.4.2 and can execute only those apps that do not contain native code or requires Android version 5.0 and above. On the other hand, CuckooDroid can be used to run on x86 based virtual machine as well as on ARM-based emulated device. CuckooDroid can be utilized and configured for Android app analysis when used with the Android version 4.4. However, for the Android version 5.0 and above, setting up CuckooDroid sandbox is not easy due to the secure boot verification in android emulator. Apart from the configuration, it can only capture the information at the framework level and fails to capture transactions happening through the lower level.

Other than the hooking and framework level API modification, virtual machine introspection (VMI) [114] techniques have also been employed for dynamic analysis like DroidScope [90], Ndroid [115]. DroidScope provides functionality to trace native instruction, dalvik instructions and taint tracking. It is built on top of the QEMU emulator and targeting the Android version 4.3 (Jelly Bean). On the other hand, Ndroid is a VMI based

taint analysis tool and can run an app that uses ART as the default runtime. However, the overhead incured by this approach makes the system more than 10X slower [90, 115].

Apart from profiling an app, analysis tools must provide the ability to hide an emulator from the running sample. However, the tools mentioned earlier also use anti-emulation-detection techniques to hide the emulator, but a smart malware author can bypass such a defense mechanisms against VM detection. For example, motion sensors can be used to evade dynamic analysis when running on emulated devices [91].

## 4.10 Discussion and Future Directions

This research presents a new dynamic analysis framework called InviSeal. It uses the emulated platform to profile an application to find malicious intentions while hiding the underneath emulated platform. To validate the efficacy of InviSeal, we use an emulation-detection library (EmuDetLib) and real malware samples from the year 2019. From the evaluation, we found that InviSeal remains undetected against the emulation-detection attacks, while other frameworks cannot hide from one or more than one attack. Now, we discuss more details of InviSeal related to the usage, impact of the rapid evolution of the Android ecosystem, application and malware, and others, followed by the future directions to extend InviSeal.

**Ability to generate realistic data for sensors:** InviSeal aims to thwart platform sensing malware that utilizes sensor data to evade dynamic analysis. To make this possible, InviSeal generates realistic sensor values while maintaining their dependencies (Figure 4.12 and Figure 4.13). To ensure proper correlation among sensors, InviSeal uses a sensor data generation algorithm (Algorithm 3). However, to generate realistic data, analysts need to define a list of sensors and their dependencies. A sensors-based emulation-detection attack can be avoided in many other ways, such as by using symbolic execution or recording the sensor data from a real device and playing it back on the emulated platform. Symbolic execution requires the interruption in the execution of a program to substitute appropriate

value by calculating it on the fly. As a result, symbolic execution slows down the system's performance and opens it up to new forms of evasion attack. The record and replay strategy is still efficient and does not incur extra overhead during analysis. However, it requires recording data for sensors after some time before replaying. Otherwise, an intelligent malware developer may use this information and arm their new malware to thwart the analysis process, which does not happen with our solution. Because in our solution, a user needs to provide two lists, one for available sensors and the other for dependencies among them. Based on these two lists, the sensor data generation algorithm generates the actual data that InviSeal feed to the emulator at runtime. Therefore, our solution does not need to change the values repeatedly. In case of any alteration in the available sensors list, like adding a new sensor or deleting an existing one, it requires changes in both lists. If dependencies among sensor are not defined properly, InviSeal fail to defend against the sensor based emulation-detection attacks.

**Impact of evolution on Android ecosystem and application:** The rapid evolution of the Android ecosystem [116] and application [117] design paradigm always imposes new challenges for application developers and security researchers. Moreover, malicious developers also arm malware with advanced techniques [32, 118] such as dynamic code loading, reflection, native code, or emulation-detection to bypass the analysis processes running on existing systems. This may also be true to some extent with InviSeal. To get around these problems, any analysis system, whether it is static or dynamic, should be able to adapt quickly to such rapid change. As InviSeal is made up of several separate modules, such as STDNeut (which runs Android OS), ARTmon, SysCallMon, and the LiME module, to profile an application across multiple layers. It can easily adapt to the rapid evolution of the Android ecosystem and applications. For example, ARTmon can work on any version of Android OS that is higher than version 7.1, as long as the Xposed framework can be used on that version. In the same way, SysCallMon will also work on another version but requires rebuilding the appropriate Android kernel and the SysCallMon module. However, STDNeut does not need any changes because it is designed using the Qemu-based Android emulator, which can run any version of Android OS.

**Impact of ARMv9's ARM Confidential Compute Architecture (CCA):** With ARMv9's ARM CCA [119] similar to SGX, an app developer can create an app that can run part of its code or the whole app in a secure environment. A secure enclave application usually does not use predefined APIs (like system calls) because doing so can reveal runtime information about the application. Since the functionality of an Android app comes from predefined APIs, InviSeal can easily extract runtime features from an app. However, let us say an Android app does not use framework layer APIs or System calls. It might be possible if all the functions are built into the app. In that case, InviSeal can not capture runtime features. Also, InviSeal can not get runtime information if the entire OS (guest OS) is running inside the secure enclave, which is possible with ARM CCA but not with SGX.

**Ability to profile collusive malware with cross-layer profiling:** One of the use cases of InviSeal is to profile/detect collusive malware that uses either single layer or multiple layers to infiltrate sensitive information in a collusive manner (Section 8.3.1). Many studies [120, 121] effectively target the detection/identification of collusive applications using static or dynamic analysis techniques. The techniques that use only static analysis [120] fail to detect collusive malware that uses both Java framework APIs and native code for collusion attacks. Furthermore, techniques based on model checking [121] by considering the execution state of the application can effectively detect collusive as compared to InviSeal. InviSeal aims to use it in multiple use cases while hiding the underlying emulated platform. It might be that InviSeal is not effective for collusive malware detection as compared to other techniques. However, it can also be used in multiple other scenarios, which is not the case with other collusive malware detection work.

**Ability to mitigate new attacks based on files and system properties:** The Android ecosystem is ever-evolving. This means that in the future, Android may change or add files and system attributes that might serve as a way of evading analysis. So, as proposed in [122], the system should be adaptable enough to include such modifications without requiring a recompile of the relevant module. In InviSeal, ARTmon provides the

defense mechanism against files and system properties-related attacks. Since ARTmon relies on external configuration files, it may be changed at any moment to protect against new threats without rebuilding the ARTmon. However, InviSeal will still be vulnerable to future evasion attempts if such updates in configuration files are not made.

**Adaptability in real devices:** InviSeal is designed to help analyst perform dynamic analysis of Android applications to find out if they are trying to do anything bad. It is mainly made for an emulated platform to stop platform-sensing malware and profile applications across multiple layers, including framework, native, and kernel. But imagine that analyst want to profile an application on a real device across multiple layers. In that case, it is also possible, given that the device has root access and the Xposed framework can be installed. Also, the analyst needs to recompile the Android kernel on the device so that SysCallMon and the LiME module can be used for system calls profiling and memory dump. There is no need to change the ARTmon in order to capture framework-level APIs.

**Adaptability for devices other than smartphones:** Some Android devices like tablets may lack cellular capabilities or do not have some sensors like GPS. In InviSeal, cellular, GPS, and other sensors fall under the sensor category. An analyst may configure InviSeal without these sensors' information to create a realistic emulated device where such sensors are not present.

### 4.10.1 Future Directions

Similar to the other analysis system, InviSeal is also having some limitations that we like to address in future. Some of the limitations and directions for future work are as follows:

**(i)** Even though InviSeal provides a strong defense against all the malware samples, it falls short if an application tries to detect Xposed framework. For example, the Snapchat application uses the native code to detect Xposed [123]. It is possible because Xposed capability is limited to the framework level API only, and here detection is performed through the native code. A more suitable defense is to handle file and system property

related emulation-detection attack at kernel-level, which is one of the future directions of this work.

**(ii)** Malware may leverage the timing channel attacks [122] to bypass the dynamic analysis process on InviSeal. Such timing channel attack includes studying the performance of the graphics sub-system, the number of instructions executed per second, or the mismatch of timing information gathered from an external server and the execution platform. For example, malware can communicate to an external NTP server to get accurate time and measure the time spent on the platform to bypass the analysis process [122]. Therefore, in the future, we like to add the capability of thwarting timing channel attacks into the InviSeal.

**(iii)** At the moment, InviSeal only dumps information about network traffic as a pcap file and does not have a module to analyze it further. There may be HTTPS/SSL traffic in the captured network information, which makes it harder to get plain text information. So, we would like to add more fine-grained network traffic monitoring utility like MITM-proxy/CharleProxy to store all the information in plain text. We also want to add a network analysis module to gather useful information from the captured traffic.

**(iv)** Like many other dynamic analysis systems, InviSeal uses the monkey [124] tool to execute a single PATH at a time. Typically, a user's activities will cause an application to change its behavior; the monkey does not have this feature. In the future, we want to enhance InviSeal with the sophisticated input-generating techniques such as Droidbot [125] to circumvent the constraints imposed by the monkey.

## 4.11 Summary

In this chapter, we have shown that anti-emulation-detection measures provided by the existing dynamic analysis frameworks are insufficient to operate in a stealthy manner resulting in detection from malware. Further, we have shown the limitations (both in terms of overhead and stealthiness) of system call monitoring utilities like `strace` to capture

the app behavior below the framework layer. To address these issues, we have presented InviSeal, a stealthy dynamic analysis framework for Android systems to perform cross-layer targeted profiling of an app in the virtual environment while hiding the underlying execution environment from the apps. We characterize the effectiveness of the InviSeal by evaluating it with different setups. We have performed experiments to showcase the effectiveness of InviSeal in providing low-overhead cross-layer profiling support with comprehensive anti-emulation-detection measures.

Furthermore, we have demonstrated the effective use of cross-layer profiling to detect colluding malware. We believe that InviSeal will improve the existing offline malware analysis system built around dynamic analysis while thwarting the emulation-detection strategies opted by the malware.

Even though InviSeal offers a strong defense from being detected as an emulated platform, malware can use timing channel attacks to detect emulated platforms (like timing measures against the graphics subsystem) to bypass the analysis process. Furthermore, Android allows the installation of an application from unverified sources (e.g., third-party market and sideloading), which opens up other ways for malware to infect smartphones. Therefore, detection of malware on a real device is essential. In the next chapter, we discuss the process of on-device malware detection and the challenges associated with designing such a system.

# Chapter 5

# DeepDetect: A Practical On-device Android Malware Detector

Even if the security system of an app store is fool-proof, it does not stop users from installing apps from untrustworthy sources, allowing malware to attack smartphones in new ways. Therefore, detection of malware on a real device is essential. In this chapter, we design a low-overhead on-device malware detector DeepDetect by employing a machine learning based model on static features. First, we discuss the challenges in designing an on-device malware detector. Later, we elaborate on the building block of the on-device malware detector, followed by the evaluation of it.

## 5.1 Introduction

In the recent years, Android has become one of the most popular operating systems (OSes) for smartphones because of its open source nature and large support for different apps. Recently, a report shared by the International Data Corporation (IDC) for smartphone

OSes showed that in the second quarter of 2021, the total market share of Android was 83.8% [4]. As a consequence of such large-scale adoption of Android, the security of these devices has become a non-trivial challenge. In the year 2019, security experts at G DATA observed that around 4.18 million new Android malware samples have been discovered [126]. This shows that a new Android malware is born every eighth second [126].

**The problem:** Malware may get unleashed into the device bypassing the defense system of Google Play Store [20] or from unverified sources (e.g., third-party market, sideloading). Therefore, on-device malware detection is crucial to stop malware affecting end-user devices. The performance of existing on-device malware detectors [6, 66] in presence of recent malware is unknown. Furthermore, these detectors utilize the API call information for malware detection that are susceptible to code obfuscation and requires significant processing time (hence impact battery life). For example, API based malware detectors take ∼11.89 seconds to extract Restricted API information which is 2.23X slower than opcode based detector (see Section 5.5.5). Moreover, most malware detectors like DroidSieve use the PRAGuard [79] dataset to evaluate their efficiency against obfuscated malware. PRAGuard dataset contains malware till March 2013. These samples are outdated and do not represent the current state of apps.

**Our goal:** We believe, an efficient and accurate on-device malware detection mechanism should complement the existing offline analysis process to stop malware infecting the end-user devices in an effective manner.

**Our approach:** To design an on-device malware detector that is faster, consumes less device energy, provides high malware detection rate and low false-positive rate, we use the following approach:

***(i)*** With code obfuscation, selection of correct features is a challenging proposition as many features either negatively impact the accuracy or unnecessarily increase the feature set size with negligible contribution towards accuracy. Therefore, we carefully select features that are either unaffected by obfuscation or we transform them into another form to make them independent of obfuscation.

***(ii)*** The most time consuming step in malware detection is the feature extraction process while other computation such as executing the trained classification model on the feature vectors require significantly less time. Hence, we design a lightweight feature extraction module that can extract features efficiently. We also adapt the N-Gram method for feature construction to preserve the relationship between opcode.

***(iii)*** With the outdated obfuscated malware samples, it is hard to measure a malware detector's efficiency with the current pace of obfuscation methods. Hence, we create a new obfuscated malware dataset that reflects the current state of obfuscation techniques and app design paradigm.

The accuracy of on-device malware detectors ([6, 66, 67, 68]) built around machine learning algorithms with static features goes down in the presence of unseen[1]/new[2]/obfuscated apps. Talos [67] uses only requested permissions to detect a malware, which can be extracted efficiently from an app by using the Package Manager (Android built-in feature). However, a malware detector based on only permissions is not a good solution because it can classify malware as benign that is obtained by introducing malicious code inside a benign app. Drebin [6], IntelliAV [66], Yuan et al. [68] includes API call information (suspicious API and/or Restricted API), which requires significant processing time (see Section 5.5.5) to extract from an app. Furthermore, the API call information is more susceptible to the code obfuscation attack, which impacts the accuracy of a model (see Section 5.5.1). Also, in the newer versions of the Android OS, some APIs may go outdated or suppressed, and the same will not be present in the future/unseen apps.

Techniques ([7, 55, 83, 84]) that are primarily proposed for deployment in the market place provides good detection accuracy for both new malware samples and obfuscated samples. One choice for on-device malware detection could be the deployment of the same model on a real device. However, these methods extract various features (permissions, intent filters, API calls, native code, etc.), and the processing time required for the extraction

[1]Samples that are not the part of training set, but they may be from the same period or prior to the samples of training set.

[2]Samples that comes after the samples used for training.

of this information is relatively high. For example, DroidSieve [7] takes an average of 2.5 seconds, while Garcia et. al. [83] takes around two seconds to analyze an app in an offline manner. Since the processing time requirement for these techniques are high on a server environment with huge processing capabilities, we speculate that the deployment of the same techniques on low-end devices will consume more time and therefore, it is not practical. Note that, Drebin takes 750 milliseconds to analyze an app on a server environment. However, when it is deployed on a real device, its reported analysis time for an app is 10 seconds. Similarly, DroidAPIMiner [55], an API based malware detector takes around 15 seconds for extracting features and 25 seconds of overall time to analyze an app in a server environment with a detection rate of ∼97%.

In this work, we present DeepDetect that enables on-device malware detection by employing machine learning on static features with significantly less processing time and device energy. Overall, our contributions are as follows:

***(i)*** We reduce the size of Dalvik opcode instruction set by combining the same semantic instructions and represent them as one instruction (Section 5.2.1). We also develop a lightweight opcode information extraction module to extract the opcode sequence from an APK inside a real device efficiently (Section 5.2.2).

***(ii)*** We develop a feature engineering framework that can drastically reduce the feature set size (from 7,12,595 to 75 features) while achieving malware detection rate of more than 97% (Section 5.3).

***(iii)*** We design an on-device malware detector (DeepDetect) that is capable of identifying a stand-alone malware (Section 5.4). We show the efficacy of DeepDetect in the presence of known[1] (training samples), unseen, and new malware in terms of detection rate (recall), precision and F1-score (Section 5.5.2). We also evaluate our model against the Pegasus malware samples that have been collected from the CloudSek organization.

[1]Samples that are the part of training data.

***(iv)*** We create a new dataset of obfuscated malware D4:Obfuscated (see Section 3.2.1) by obfuscating 4,993 unique malware with six different categories. We evaluate DeepDetect against these obfuscated samples, DeepDetect correctly detects 95.57% malware, which is an overall drop of 1.55% compared to the same set of non-obfuscated samples (Section 5.5.3). Additionally, we evaluate DeepDetect against a new dataset D5:Biases-Free (spanning over four years 2016 to 2019, see Section 3.2.5) downloaded from Androzoo and eliminated both the potential biases i.e. spatial and temporal. In this evaluation, DeepDetect is able to detect ∼97% of malware while generating ∼1.4% false alarms (Section 5.5.4). Further, we show the runtime performance in extraction of different features on a real device in terms of the execution time and device energy consumption. The opcode based malware detector is 2.23X faster than the Restricted API based detector and consumes 2.17X less device energy (Section 5.5.5).

## 5.2 Feature Extraction

Feature extraction is an important step in machine learning based malware detection systems. In this section, we discuss about the type of features extracted from the dataset along with the process of feature extraction.

### 5.2.1 Type of Features

In this work, we extract features from two locations, i.e., (i) Application Manifest file [19] and (ii) Dex code. The Manifest file contains information about the Android app, whereas the Dex code holds the main execution logic. Our primary goal is to design a malware detector based on the static features while ensuring that the prediction accuracy is not affected due to the obfuscation. Note that, here obfuscation includes all defense mechanisms like packed malware, dynamic code loading, native code, and others [32]. Generally, the obfuscation is applied to the code available in the Dex file, where a malware writer hides the use of actual API by transforming it into another form. The extraction of used API

information from the Dex file may lead to miss classification, and the overall performance of the detection model may be negatively impacted. Therefore, we utilize low-level Dalvik bytecode (opcode) instead of the high-level API information and the features from the Manifest file to design an obfuscation-resilient malware detector.

**Features from the Manifest file:** There are four categories of features [6] that can be extracted from the Manifest file. The description of the features are as follows:

***(i)* Requested permissions**: An app has to request permission to access important and sensitive information. Malicious apps request certain permissions more frequently than benign apps.

**(ii) App components**: Generally, an app contains four types of components. Each component either defines user interfaces or interfaces to the system. These components include activities, content providers, services, and broadcast receivers.

**(iii) Intent filters**: Using intent, inter-process and intra-process communication is performed. Malware often listens to such intents.

**(iv) Hardware components**: Access to a certain hardware may have some security implications.

**Features from the Dex code:** API call information extracted from the Dex code is prone to obfuscation attacks. As an API is treated as a string by the static analyser, a malware can bypass it very easily because the execution happens with a stream of Dalvik opcodes. Dalvik instruction set contains 230 instructions to perform a designated task. These instructions include method call instructions, branch instructions, data manipulation instructions, and others. We generate a 2-Gram (N-Gram [127]) sequence of Dalvik opcodes for each function to use them as features for malware detection. N-Gram is widely used in natural language processing, and it has also been adapted for malware analysis.

As the Dalvik instruction set is large enough (230 instructions), a possible number of unique features obtained using N-Gram is approx $230^N$. Such a large number of features

are not suitable to design an on-device malware detector. Therefore, we have clubbed several instructions to reduce them into one instruction based on their usages (Table 5.1), like move instructions or method call instructions similar to the approach used in [128, 53]. However, our reduced instruction set contains 17 instructions (Table 5.1), which is slightly more than the reduced instructions set of TinyDroid [53] and Dong et al. [128]. Our reduced instruction set is based on the 228 instructions (excluding the NOP and empty method call instruction), whereas the simplified instruction set of TinyDroid and Dong et al. includes only 107 and 218 instructions, respectively. Apart from the size of the instruction set, we also include features from the Manifest file and reduce the feature set size through the feature engineering module which is different from the Dong et al. and TinyDroid.

TABLE 5.1: Reduced instruction set with description.

| **Symbol** | **Description** |
|---|---|
| A | Arithmetic operation instructions |
| B | Branch instruction (Conditional jump like if-eq) |
| C | Comparison instruction like cmpl-float |
| D | Data Definition instructions like const/4 |
| F | Type conversion instructions (int-to-long, int-to-float) |
| G | Get instructions (aget, aget-wide) |
| I | Method call instructions (invoke-direct, invoke-virtual ...) |
| J | Jump instructions (Unconditional) like goto |
| L | Lock instruction, use to acquire/release a lock (monitor-enter and monitor-exit) |
| M | Data manipulation instruction like move and its variants |
| O | Exception instruction (through) |
| P | Put instructions (aput, aput-wide) |
| R | Return instruction like return-void |
| S | Bit-wise operation instructions (and-int, shl-int) |
| T | Type judgement like check-cast |
| V | Array operation instructions like array-length |
| X | Switch case instructions |

The above mentioned features are extracted using Androguard [33] and represented as strings. From each sample, we have extracted all the information discussed above as features. In Table 5.2, column named **Original** shows the number of unique features extracted from the D1:Base-Dataset (see Section 3.2.1). There is a total of 7,12,595 unique features extracted from the Manifest file and Dex code. Using such a large number of features for an on-device malware detection leads to significant overhead in terms of processing time and computation requirements. To overcome this limitation, we need to reduce the

TABLE 5.2: Effect of feature selection & encoding on extracted features.

| **Category** | **#Features** | |
|---|---|---|
| | **Original** | **Encoding** |
| Requested Permission | 23,175 | 668 |
| Hardware Component | 245 | 245 |
| Intent Filters | 50,257 | 1 |
| Activities | 5,24,989 | 1 |
| Services | 57,202 | 1 |
| Broadcast Receivers | 49,751 | 1 |
| Content Providers | 6,659 | 1 |
| Custom Permissions | 0 | 1 |
| 2-Gram Opcode Sequence (2-Opc) | 317 | 317 |
| **Total Features** | **7,12,595** | **1,236** |

dimension of the feature space to build an efficient on-device malware detector. We discuss the process involved in reducing the feature set size in Section 5.3. Next we discuss the efficient feature extraction method designed for a real device.

### 5.2.2 On-device Efficient Feature Extraction:

We use the features from two locations as discussed in Section 5.2.1. To extract features from the Manifest file, Android provides an in-built functionality called Package Manager (referred to as PM). Whenever an app gets installed (or gets updated, which is done frequently) on an Android device, the PM maps all the information related to the Manifest file. Hence, we use PM directly to extract features from the Manifest file efficiently. However, Android does not provide any in-built functionality to extract information (opcode) from the Dex files of an APK. Hence, we have designed an efficient and lightweight opcode information extractor with the help of *DexLib2* library.

The *DexLib2* is a Java library to process the Dalvik executable code, which has been used by many heavy APK processing frameworks like APKTool to perform reverse engineering. One can argue that, if APKTool is available, then why a new feature extraction method is needed for on-device? Why can we not use APKTool directly? The reason is, APKTool disassembles an APK and dumps the disassembled code into the secondary storage in smali code (see listing 5.1, a smali code snippet of a method disassembled using APKTool). The generated smali code is then used to extract the features by parsing them, requiring a lot

of string processing/comparison and file system operation. Both file operation and string comparison requires significant processing capability and time.

```
const/4 v0, 5
const/16 v1, 10
add-int v2, v0, v1
invoke-virtual {v3, v2}, Activity;->setter(I)V
return-void
```

LISTING 5.1: Sequence of Dalvik opcodes (smali code).

To extract opcode features efficiently, we deal with the Dex code as follows:

***(i)*** Dex file is a stream of Hex code. Hence, we operate directly on the sequence of hex stream (avoiding string operations).

***(ii)*** We read Dex files one-by-one (in case of MultiDex app) from an APK and operate only in-memory (avoiding file system operation).

Listing 5.2 shows the Dalvik opcode sequence in the hex stream (little-endian format), equivalent to the smali code shown in listing 5.1. Compared to the smali code (listing 5.1), the hex stream (listing 5.2) does not contain any string. Hence, it does not require time-consuming string operations to extract opcode information (highlighted in blue color in both listings 5.1 and 5.2).

```
1250
1302 0a00
9002 0001
6e20 cc00 2300
0e00
```

LISTING 5.2: Sequence of Dalvik opcodes (hex code).

## 5.3 Feature Engineering

A model based on collecting as much data as possible enables the classifier to learn more. However, the computational power and processing time necessary to construct such models

are immense. Such models cannot be used on a real device in order to detect malware. Therefore, we require a mechanism that would allow us to train a machine learning model with fewer features while preserving the detection accuracy.

Often most of the features are usually irrelevant or redundant and increase the model complexity. Therefore, we consider only the essential and relevant features. Feature selection/reduction methods are often used to solve such problems. Figure 5.1 shows the flow of the feature engineering module. This module has three major phases, which are discussed in subsequent sub-sections.

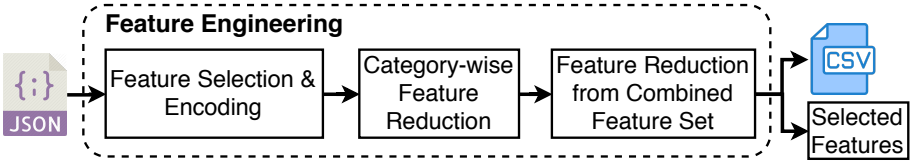



FIGURE 5.1: Flow of feature engineering module.

### 5.3.1 Feature Selection and Encoding

Android app developers can provide any name to custom permissions, services, activities, and other user-defined entities/components. Such user-defined components have a massive number of features due to user-defined names. Despite having a large number of binary-valued features in each set for components, they do not show good prediction abilities concerning the number of features available in each feature set. Hence, in the quest of reducing the feature set, one should carefully handle loss of information. Therefore, instead of eliminating these binary-valued features, we transform them by maintaining a count of the components in each category. After the transformation, each of these set holds the frequency of their usage for an app. Another change can take place in requested permissions where we keep Android defined permissions as binary-valued features and make the count of remaining permissions (custom permission). Note that, we did not perform any transformation to the features extracted from the Dex code. As it is already done by reducing the instructions as discussed in Section 5.2.1. We only use frequency of 2-Gram opcode sequence extracted from every function inside the Dex code.

Finally, after transformation and encoding of the extracted features, we have feature sets that contain either binary value or the frequency of their usage. The **Encoding** column in Table 5.2 shows the resulting set of features after this step.

### 5.3.2 Category-wise Feature Reduction

Most of the categorical sets individually have a lot of predictive powers. Hence, we optimize the performance of each of the binary-valued feature set and 2-Gram opcode sequence, individually and then combine their predictive power to build a better model.

Category-wise feature reduction involves two processes for feature reduction, as shown in Figure 5.2. Both the binary-valued feature sets (see Section 5.3.1) and opcode sequence are passed through each of these processes to filter out the redundant and irrelevant features in each category. All of the steps involved in reducing features by category are explained below.

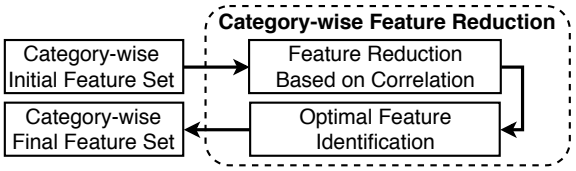



FIGURE 5.2: Category-wise feature reduction process.

**Feature Reduction Based on Correlation:** Often a dataset contains some features that are highly correlated with each other and provides the same information. Keeping all such features (correlated features) increases the complexity of the model without contributing towards classification efficiency. This step addresses the issue by finding all the correlated features. Correlation between two features is performed based on the Pearson's correlation coefficient [129] that lies between $-1$ to 1. Pearson value closer to 0 denotes weak correlation whereas value closer to 1 and $-1$ implies strong positive and negative correlations, respectively. In both the strong correlations (i.e. positive and negative), it is possible to minimize the feature set size without compromising the models' performance by using only one feature.

TABLE 5.3: Terminology used in feature engineering.

| Term | Description |
|---|---|
| $COR_t$ | Threshold for eliminating the correlated values. |
| $Acc$ | It represents the accuracy of a detection model. |
| $Acc_R$ | Highest accuracy given by the RFECV [130]. |
| $Feat_R$ | Optimal #features used by RFECV to achieve accuracy $Acc_R$. |
| $RFE_t$ | It is a penalty over the highest accuracy $Acc_R$ while selecting less #feature in place of optimal #feature $Feat_R$. |
| $Feat_C$ | Chosen #Feature with penalty $RFE_t$ over $Acc_R$. |
| $Acc_C$ | Accuracy achieved while taking only $Feat_C$ features. |
| $Pre$ | Denotes the precision at which detection model can operate. |
| $Rec$ | Denotes the malware detection rate (recall) of a detection model. |

By this analogy, we apply different Pearson correlation values as a threshold in both the directions (i.e., negative and positive) to filter out highly correlated features. The effect of correlation value (referred to as $COR_t$[1]) for different threshold (0.5 to 1.0) against the accuracy[2] of the detection model has been shown in Table 5.4. The $COR_t$ value 1.0 represents the original features without any reduction process. The results shown in Table 5.4 are obtained by evaluating a RandomForest model on the training set (see Section 5.5) with 10-fold cross-validation. We use a threshold of 0.8 for requested permissions and hardware components while 0.9 for 2-Gram opcode (highlighted in Table 5.4) as it results in the correct tradeoff between accuracy and number of features (significant reduction in feature set size with minimum loss in accuracy). Note that, we use similar methods (using training set) for the remaining stages of feature engineering.

TABLE 5.4: Effect of Pearson coefficient threshold ($COR_t$) on Accuracy (Acc) and #Features (#Feat).

| $\mathbf{COR_t}$ | **Acc(%) / #Feat** | | |
|---|---|---|---|
| | **ReqP** | **HWC** | **2-Opc** |
| 0.5 | 93.69 / 533 | 60.58 / 194 | 86.32 / 39 |
| 0.6 | 94.00 / 562 | 60.79 / 207 | 90.05 / 55 |
| 0.7 | 93.99 / 575 | 60.82 / 212 | 94.82 / 72 |
| **0.8** | **94.74 / 602** | **60.80 / 224** | 95.50 / 104 |
| **0.9** | 94.86 / 626 | 60.79 / 226 | **95.99 / 172** |
| 1.0 | 94.89 / 668 | 60.85 / 245 | 96.28 / 317 |

*Note:* ReqP=Requested Permissions set, HWC=Hardware Component set, 2-Opc=2-Gram opcode sequence

**Optimal Feature Identification:** This process utilizes RFECV (recursive feature elimination with cross validation) [130] to identify the optimal feature set. RFECV uses a

[1]Summary of notations in Table 5.3 used in the remaining sections.
[2]Accuracy represents the number of samples (in percentage) correctly classified.

feature ranking method and selects the best features that contributes significantly in solving the desired problem. We provide the classifier $C$ as RandomForest, ranking function $F$ as accuracy and the number of features $N$ (set to 1) as input to RFECV for feature elimination. As a result, RFECV provides a grid of score and the set of optimal features that gives highest accuracy (see Figure 5.3). Corresponding to requested permissions, hardware components and 2-Gram opcode sequence, the optimal number of features selected with highest accuracy by the RFECV is 371, 185 and 169, respectively, which are still a lot of features. *However, if we carefully observe Figure 5.3, we find that with a significantly less number of features result in an accuracy very close to the maximum achievable accuracy.* With this observation, we define a threshold $RFE_t$, which is a penalty in choosing less number of features in terms of accuracy.

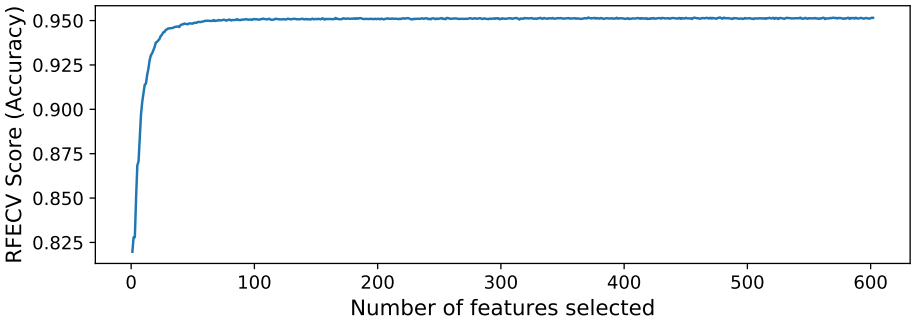



FIGURE 5.3: Optimal #features Vs accuracy graphs for requested permissions.

For example, let the highest accuracy given by RFECV is $Acc_R$ with optimal number of features $Feat_R$. However, we observe $Feat_R$ is still large and can be further reduced finding a sweet spots without significantly compromising on accuracy. Let the chosen accuracy from RFECV grid score be denoted by $Acc_C$ and the corresponding number of features as $Feat_C$. Then the relation between the threshold $RFE_t$, highest accuracy $Acc_R$ and the chosen accuracy $Acc_C$ is shown in Equation 5.1.

$$Acc_R - Acc_C \leq RFE_t \quad \text{and} \quad Feat_C < Feat_R \tag{5.1}$$

The effect of the $RFE_t$ with varying values from 0.0 to 0.5 is shown in Table 5.5 where 0.0 denotes the RFECV score with highest accuracy and the threshold is the difference

between $Acc_R$ and $Acc_C$ in percentage. As shown in Table 5.5, the changes in threshold $RFE_t$ significantly reduces the feature set size while maintaining acceptable accuracy score. In this step, we have selected 0.5 as the $RFE_t$ value where a drastic reduction in the feature set size can be observed. The remaining relevant features contributes effectively towards solving the desired problem. However, we do not know the features involved to achieve the same results except the count of those features. To extract the required features, we have used Recursive Feature Elimination (RFE) [131] that takes a classifier and the number of features we want to select (as information retrieved using RFECV and threshold $RFE_t$) as input to obtain the list of features without impacting the accuracy.

TABLE 5.5: Effect of $RFE_t$ threshold on Accuracy (Acc) and #features.

| $\mathbf{RFE_t}$ | **Acc(%) / #Features** | | |
|---|---|---|---|
| | **ReqP** | **HWC** | **2-Opc** |
| 0.0 | 94.77 / 371 | 60.79 / 185 | 96.01 / 169 |
| 0.1 | 94.76 / 86 | 60.75 / 17 | 95.92 / 69 |
| 0.2 | 94.68 / 60 | 60.72 / 13 | 95.90 / 62 |
| 0.3 | 94.57 / 52 | 60.65 / 13 | 95.76 / 48 |
| 0.4 | 94.47 / 41 | 60.62 / 13 | 95.76 / 37 |
| **0.5** | **94.34 / 40** | **60.60 / 12** | **95.64 / 30** |

*Note:* ReqP=Requested Permissions set, HWC=Hardware Component set, 2-Opc=2-Gram opcode sequence

### 5.3.3 Feature Reduction from Combined Feature Set.

In Section 5.3.2, we have performed category-wise feature reduction where binary features from two categories (requested permissions and hardware components) along with the frequency of 2-Gram opcode sequence are involved. However, the original feature set contains three types of features viz. binary features, 2-Gram opcode sequence and the numeric features. This section combines all features and performs a feature reduction in the combined feature set. This process includes two steps — *(i)* combining the features and selecting the best combination of feature set and *(ii)* feature reduction on the selected combined feature set.

**Combining Feature Set:** In Section 5.3.1, we have selected obfuscation resilient features divided in four sets—one set corresponding to numerical features, one related to the opcode sequence and the remaining two sets related to the binary features. Using four different

feature sets, we combine all the features in different combinations and select the best combination among them. In total, there are 11 unique combinations, which are possible for four sets. We have trained a RandomForest classifier on these combinations and the result for them is summarized in Table 5.6 where performance of all the combinations are enlisted in three evaluation metrices—*(i)* accuracy (Acc), *(ii)* precision[1] (Pre), and *(iii)* recall[2] (Rec). As shown in Table 5.6, the hardware components feature set do not contribute towards accuracy in a significant manner; neither individually nor by combining with other features. If we compare detection results with two different combinations with high accuracy—*(i)* numeric feature combined with requested permissions & opcode sequence (76 features), and *(ii)* combining requested permission with opcode sequence only (70 features), the results indicates that the performance of both the sets are almost same but differs in the number of features. One could simply select the feature set which contains less features in this case combination *(ii)*. However, in place of combination *(ii)*, we select combination *(i)*, because the contribution of numeric feature set is significant when it is combined with the requested permissions (see Table 5.6). Therefore, we select combination of Numeric feature, requested permissions and opcode sequence for the next phase of reduction. Note that, here selection of combination is based on the analyst point of view as both combination perform almost equally.

TABLE 5.6: Effect of combining different feature set.

| **Combination** | **#Features** | **Acc (%)** | **Pre (%)** | **Rec (%)** |
|---|---|---|---|---|
| Num+HC | 18 | 86.45 | 86.55 | 85.46 |
| Num+OP | 36 | 96.90 | 96.93 | 96.90 |
| HC+OP | 42 | 96.07 | 96.19 | 96.07 |
| Num+RP | 46 | 96.45 | 96.46 | 96.24 |
| Num+HC+OP | 48 | 96.87 | 96.89 | 96.87 |
| RP+HC | 52 | 95.16 | 95.08 | 94.96 |
| Num+RP+HC | 58 | 96.57 | 96.59 | 96.37 |
| **RP+OP** | **70** | **98.15** | **98.15** | **98.15** |
| **Num+RP+OP** | **76** | **98.14** | **98.15** | **98.14** |
| RP+HC+OP | 82 | 98.12 | 98.12 | 98.12 |
| Num+RP+HC+OP | 88 | 98.12 | 98.12 | 98.12 |

*Note:* RP=Reduced Requested Permissions set, HC=Reduced Hardware Component set, Num=Numeric Feature (features for that we have taken frequency of their usage except n-Gram features), OP=Reduced 2-Gram Opcode Sequence, Acc=Accuracy, Pre=Precision, Rec=Recall

[1]Precision denotes the fraction of malware correctly detected.
[2]Recall represents the malware detection rate for a model.

**Feature Reduction in Selected Feature Set:** In this step, we perform final optimization on the feature set of the selected combination. The aim of this optimization is to eliminate some features that increases the processing time and requires extra support for extraction. In the selected feature set, such features are Intent Filter (referred to as I) and Custom Permissions (referred to as C). We observe the effect on accuracy, precision, and recall (see Table 5.7) when eliminating either one or both features from the feature set. From Table 5.7, we find that the elimination of Custom Permissions (C) significantly impacts the accuracy of malware detection (recall). On the other hand, the elimination of the Intent Filter does not affect the detection rate of the model. After exclusion of Intent Filter, the resulted feature set (of size 75) is used to learn the final detection model.

TABLE 5.7: Elimination of feature from combined feature set.

| **Feature Set** | **#Feature** | **Acc (%)** | **Pre (%)** | **Rec (%)** |
|---|---|---|---|---|
| Num+RP+OP | 76 | 98.14 | 98.15 | 98.14 |
| **Num+RP+OP-I** | **75** | **98.18** | **98.18** | **98.18** |
| Num+RP+OP-C | 75 | 98.08 | 98.08 | 98.08 |
| Num+RP+OP-I-C | 74 | 98.13 | 98.13 | 98.13 |

*Note:* RP=Reduced Requested Permissions set, Num=Numeric Feature, OP=Reduced 2-Gram Opcode Sequence, I=Intents Filter, C=Custom Permissions, Acc=Accuracy, Pre=Precision, Rec=Recall

## 5.4 DeepDetect: Building the System

So far, we have extracted the features from the dataset (see Section 5.2) and selected the most relevant features (see Section 5.3) to design an on-device malware detector. In this section, we first provide an overview of DeepDetect followed by the off-device training process of the machine learning model and porting it for mobile devices to detect malware on real devices.

### 5.4.1 Overview

In DeepDetect, we train a machine learning model on a server machine. First, we extract the static features from the Manifest file and Dex code (see Section 5.2). Then we pass these features to the feature engineering process. In feature engineering (see Section 5.3),

we first eliminate the effect of obfuscation with the help of transformation and then reduce the size of feature dimension by applying a multilevel feature selection/reduction process. At last, a detection model is learned and embedded into an app for on-device detection. We describe learning the malware detection model and detecting malware on a real device in Section 5.4.2 and Section 5.4.3, respectively.

### 5.4.2 Learning Model

The final feature set obtained from Section 5.3.3 contains obfuscation-resilient features, and obfuscation techniques used by the malware writer cannot affect the prediction ability of a model built upon them. As our primary goal is to detect malware on-device, the classifier's choice for the final detection model should be lightweight. Keeping this in mind, we use `TensorFlow` [132] library (developed by Google) to build the final model. Google also provides a light version of `TensorFlow`, i.e., `TensorFlow Lite`, which is designed for mobile devices. We use TensorForest [133] to learn final detection model. We use training set for learning the final model and serialize the model into a file. The saved model occupies 869 KB of space in the system. The saved model needs to be converted into the `.tflite` format for direct use with `TensorFlow Lite`. The `TensorFlow Lite` converter has been used to convert the learned output so that we can use it in an Android device to detect malware seamlessly. At last, we obtain a final `TensorFlow Lite` model of size ∼150 KB only. The final detection model is provided to the DeepDetect along with the features used in training to detect malware on-device.

### 5.4.3 On-device Detection

To detect malware on a real device, we require feature vector from an APK, so that we can pass it to the already trained model (see Section 5.4.2) for the detection result. The procedure for extracting the feature from a list of app(s) (single app as well as multiple apps) and embedding them into the feature vector is shown in Algorithm 5 by utilizing the Package Manager (referred to as $PM$) and customised opcode information extraction

**Algorithm 5:** Feature Vector Generation

**Input :** $List_{pkgs}$, $Model_{perm}$, $Model_{2-opc}$
**Output:** $Vector_{feat}$

1 $Vector_{feat} \leftarrow \phi$ // Vector of features
2 $N_{act} \leftarrow ActivitiesCount(PM, List_{pkgs})$
3 $N_{serv} \leftarrow ServicesCount(PM, List_{pkgs})$
4 $N_{recv} \leftarrow ReceversCount(PM, List_{pkgs})$
5 $N_{prov} \leftarrow ProvidersCount(PM, List_{pkgs})$
6 $List_{perm} \leftarrow Permissions(PM, List_{pkgs})$
7 $Freq_{2-opc} \leftarrow GetTwoGramOpcodeFreq(List_{pkgs})$
8 $N_{CustPerm} \leftarrow CustPermCount(List_{perm})$
9 $append(Vector_{feat}, (N_{act}, N_{serv}, N_{recv}, N_{prov}, N_{CustPerm}))$
10 **foreach** *perm in* $Model_{perm}$ **do**
11  **if** *perm is in* $List_{perm}$ **then**
12  $bit \leftarrow 1$
13  **else**
14  $bit \leftarrow 0$
15  $append(Vector_{feat}, bit)$
16 **foreach** *twoOP in* $Model_{2-opc}$ **do**
17  **if** *twoOP is in* $Freq_{2-opc}$ **then**
18  $count \leftarrow get(Freq_{2-opc}, twoOP)$
19  **else**
20  $count \leftarrow 0$
21 **return** $Vector_{feat}$

module (referred to as *GetTwoGramOpcodeFreq* designed using the process discussed in Section 5.2.2).

Algorithm 5 takes the list of the apps in terms of their package names (referred to as $List_{pkgs}$), list of requested permission (referred to as $Model_{perm}$) and the list of selected 2-gram opcode sequence (referred to as $Model_{2-opc}$) as input. In Algorithm 5, features are extracted from apps by querying PM and custom opcode information extractor (line 2 to 7), number of custom permissions are filtered (line 8), and all features are encoded into the feature vector (line 9 to 20). Note that, requested permissions that do not start with "android.permission" are called custom permissions. The feature vector obtained using Algorithm 5 is then passed to the detection model (see Section 5.4.2). The detection model analyses the extracted features and provides the result in terms of a binary answer

where a `true` implies malware and a `false` implies benign. If the targeted app is flagged as malware, the same is notified to the user and provides an option to uninstall the app.

## 5.5 Evaluation

To evaluate the effectiveness of DeepDetect in terms of malware detection accuracy and runtime performance, we have separated 20% samples (referred to as evaluation set) from the D1:Base-Dataset described in Section 3.2.1 and assumed them to be unseen apps. In all the experiments, we train every model using the remaining 80% samples (referred to as training set) of the dataset, which has samples till April 2018. *Note: only runtime performance experiments (Section 5.5.5) are performed on real smartphones to measure execution time and device energy consumption. Rest of the experiments are performed on a server machine.* This section answers the following research question to evaluate the proposed detection system effectively:

***(i)* Robustness against known, unseen, and new samples** (Section 5.5.2): Does DeepDetect maintains its prediction capability against known, unseen, and new samples?

***(ii)* Impact of obfuscation** (Section 5.5.3): What is the impact of obfuscation on the malware detection rate?

***(iii)* Effect of experimental biases** (Section 5.5.4): How DeepDetect performs after eliminating potential experimental biases?

***(iv)* Runtime overhead** (Section 5.5.5): Does 2-Gram opcode sequence features consume less device energy as compared to other features from Dex code?

### 5.5.1 Performance Comparison of Features

In this experiment, we extract seven categories of features from the Dex code—*(i)* 1-Gram sequence of opcode, *(ii)* 2-Gram, *(iii)* 3-Gram, *(iv)* Used Permissions (UP), *(v)* Suspicious

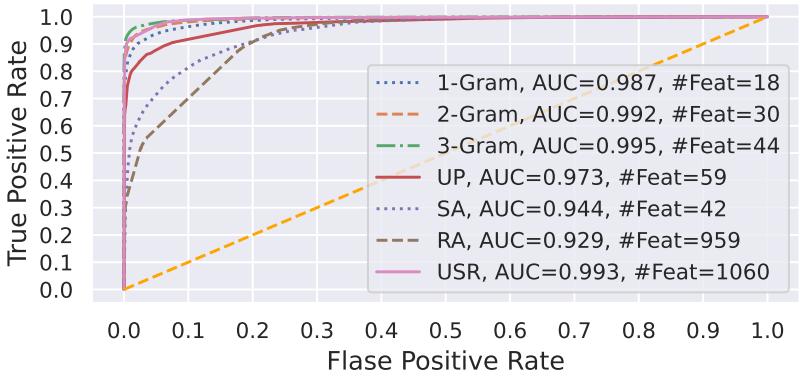



FIGURE 5.4: AUC-ROC curve for the model build on various feature extracted from Dex file. UP: Used Permission, SA: Sensitive APIs, RA: Restricted APIs and USR: Combined features (UP, SA and RA). #Feat: Number of Features.

APIs (SA), *(vi)* Restricted APIs (RA), and *(vii)* combined features from UP, SA, RA (referred to as USR). To compare performance of these feature sets, we train a RandomForest model on the training set and evaluated against the evaluation set. Figure 5.4 shows the Receiver Operating Characteristic (ROC) curves obtained from the evaluation results of the trained model along with the value of AUC (Area Under the Curve) and the number of features (#Feat) in a feature set. ROC curve represents the relationship between the true positive rate (TPR), and false-positive rate (FPR). The AUC measures the ability of a classifier (model) to distinguish between classes and is generally used as a summary of the ROC curve. *The higher the AUC, the better the model's performance at distinguishing between the positive and negative classes.* As shown in Figure 5.4, high AUC scores are observed for 2-Gram, 3-Gram, and USR (more than 99%). However, the number of features in USR is relatively large as compared to 2-Gram and 3-Gram. Also, the device energy consumption and time required to extract features are also large (see Section 5.5.5). Hence, USR is not a good choice of features to design an on-device malware detector. Therefore, from the remaining two feature sets, any one can be used for on-device malware detection. However, we select 2-Gram because it consumes ∼1.4X less device energy as compared to 3-Gram. *Note: The main aim of this experiment (and experiment in Section 5.5.5) is to select best features that can be efficiently extracted from the Dex code and combined with features extracted from the Manifest file.*

TABLE 5.8: Evaluation of final model with known, unseen, new and Pegasus samples.

| **Dataset** | **Pre (%)** | **Rec (%)** | **F1 (%)** | **FPR (%)** |
|---|---|---|---|---|
| Training Set | 99.98 | 99.95 | 99.95 | 0.01 |
| Evaluation Set | 98.05 | 97.50 | 97.69 | 1.51 |
| D2:AndroZoo-2019 | 97.70 | 97.12 | 97.69 | 1.73 |
| D3:Pegasus (5 Sample) | – | 100 | – | – |

### 5.5.2 Performance Against Known, Unseen, and New Samples

To show the effectiveness of DeepDetect in identifying unseen samples, we use the evaluation set. We use D2:AndroZoo-2019 (see Section 3.2.2) dataset as new samples because these samples are born after the training samples. Apart from the D2:AndroZoo-2019 dataset, we also used D3:Pegasus (see Section 3.2.3) dataset to evaluate DeepDetect against Pegasus malware. For the known samples, we have used the same samples used for training the final model. Table 5.8 summarizes the evaluation results for the known (training set), unseen (evaluation set), new (D2:AndroZoo-2019), and Pegasus malware samples (D3:Pegasus) with four evaluation metrices–*(i)* F1-score[1] (referred to as F1), *(ii)* precision, *(iii)* recall, and *(iv)* false-positive rate (referred to as FPR). When the model is evaluated against the known samples, the model correctly classifies 99.90% malware and generates 0.01% of false alarms. For the unseen samples, our detection model correctly detects 97.50% malware with a false positive rate of 1.51%. In the presence of new samples, our model detects 97.12% of new malware while generating 1.73% of false alarms. Interestingly, our model is able to detect all the Pegasus malware samples. Maybe this is possible because Pegasus samples are pre-2019. In comparison to the state-of-the-art on-device detector (Drebin [6]), DeepDetect detects ∼3.5% more malware (unseen malware) using only 75 features, whereas Drebin uses 0.5 million features with a malware detection rate of ∼94%. If we compare DeepDetect with the recent on-device malware detector IntelliAV [66], DeepDetect outperforms in both type of samples, i.e., known and unseen/new samples. DeepDetect's malware detection rate in case of known samples is 0.15% more as compare to IntelliAV. In the presence of unseen/new samples, the detection rate of IntelliAV is still less than the best state-of-the-art detector Drebin. Further, we have analysed the importance of features in differentiating a malware from the benign. In

[1]F1-score represents the weighted average of recall and precision.

our analysis we found that GET_ACCOUNT requested permission and SB (Bit-wise operation followed by branch instruction) opcode sequence play a major role. For more details about the list of features, their importance and more experiment (with other classifiers and performance metrics), please refer to the Appendix A.

### 5.5.3 Evaluation Against Obfuscated Malware

To evaluate our model against the obfuscated samples, we use D4:Obfuscated dataset which contains 4,993 obfuscated samples in six categories. The number of obfuscated samples in each category and their evaluation with our detection model are shown in Table 5.9. We have also evaluated the same non-obfuscated samples in each category against our detection model. The number of samples detected in non-obfuscated and obfuscated samples are shown in the **Original** and **Obfuscated** columns of Table 5.9, respectively. *Note: We have trained our model on the training set only, which contains samples till April 2018.*

TABLE 5.9: Evaluation of final model against obfuscated malware.

| **Category** | **#Samples** | **#Sample Detected** | | **Detection rate drop (%)** |
|---|---|---|---|---|
| | | **Original** | **Obfuscated** | |
| trivial | 160 | 156 | 156 | 0 |
| renaming | 570 | 554 | 554 | 0 |
| encryption | 1,135 | 1,102 | 1,096 | 0.53 |
| reflection | 252 | 241 | 239 | 0.79 |
| code | 2,429 | 2,358 | 2,298 | 2.47 |
| mix | 447 | 438 | 429 | 2.01 |
| **Overall** | **4,993** | **4,849** | **4,772** | **1.55** |

The results shown in Table 5.9 indicate that the detection rate of DeepDetect does not go down for trivial and renaming obfuscation techniques. The main reason behind this is the feature encoding, where we take count of user-defined similar entities like activities. However, we observe some drop in detection rate for other categories. We see a maximum drop in detection rate (up to 2.47%) for code obfuscation techniques. In general, our detection model can detect 95.57% of malware, whereas, for the same non-obfuscated malware sample, it achieves a 97.12% of detection rate. Therefore, the overall drop in malware detection rate in the presence of obfuscated samples is 1.55%.

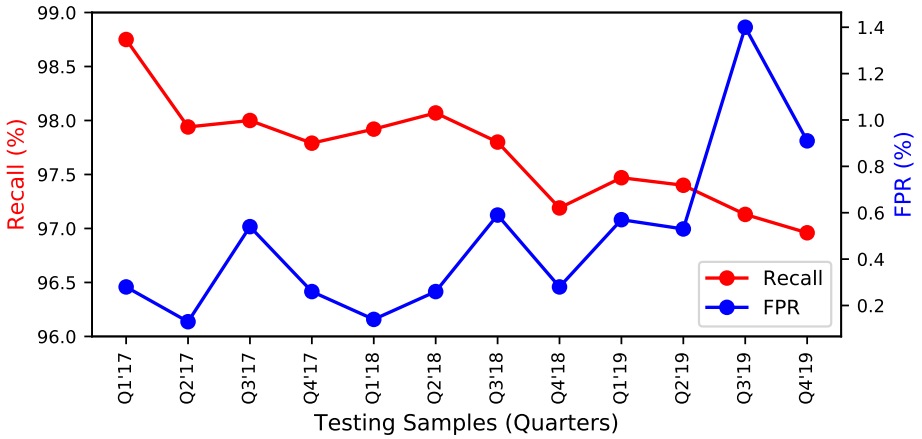



FIGURE 5.5: Detection results after removing experimental bias (Space and Time).

### 5.5.4 Evaluation After Elimination of Experimental Biases Across Space and Time

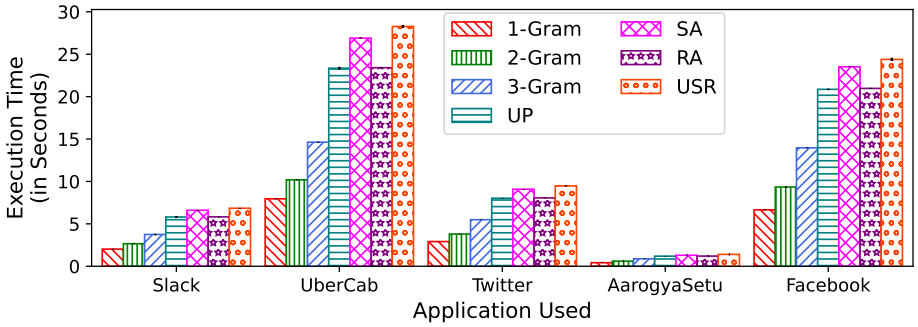
TESSERACT [72] has shown that the experiment done to evaluate existing Android malware detectors have two potential experimental biases–*(i)* Spatial bias and *(ii)* Temporal bias. Spatial bias occurs due to the incorrect distribution of malware and benign samples in the dataset, whereas temporal bias is caused by the incorrect time splits of training and testing samples. DeepDetect evaluation is free from the temporal bias when evaluated against D2:AndroZoo-2019 and Obfuscated samples, but spatial bias is still present. Therefore, we evaluate DeepDetect against D5:Biases-Free dataset which is free from both the biases and contains 87,632 (with ∼10% malware) unique samples spanning for four years (2016 to 2019). We train the DeepDetect model on the sample from year 2016 and test it quarter-wise against the years 2017, 2018, and 2019. Samples in each quarter contain ∼10% malware, and the remaining are benign. The evaluation result (see Figure 5.5) shows that DeepDetect can detect ∼97% of malware while generating ∼1.4% false alarms when evaluated after elimination of experimental biases across space and time. It indicates that our model is also robust against experimental biases. However, when TESSERACT evaluated Drebin and MaMaDroid [71] against potential biases, the performance decreased up to 50%.

TABLE 5.10: Android apps used for runtime performance and device energy consumption.

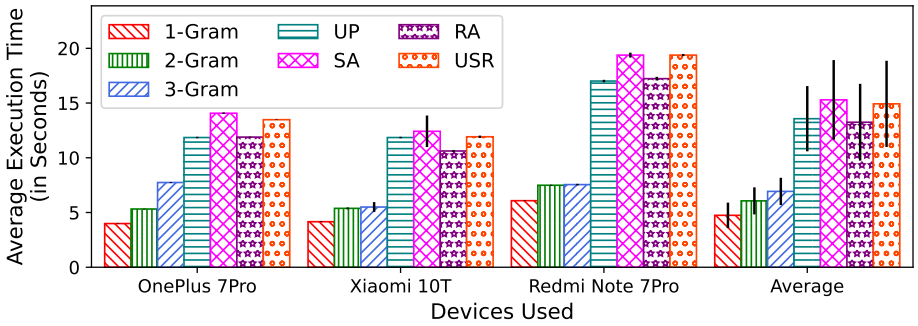
| App Name | #Dex Files | Size (MBs) | |
|---|---|---|---|
| | | APK | Dex Files |
| Slack | 3 | 61 | 24.6 |
| UberCab | 15 | 50 | 116.6 |
| Twitter | 5 | 19 | 33.6 |
| AarogyaSetu | 1 | 4.3 | 5 |
| Facebook | 11 | 55 | 81.2 |

### 5.5.5 Runtime Efficiency

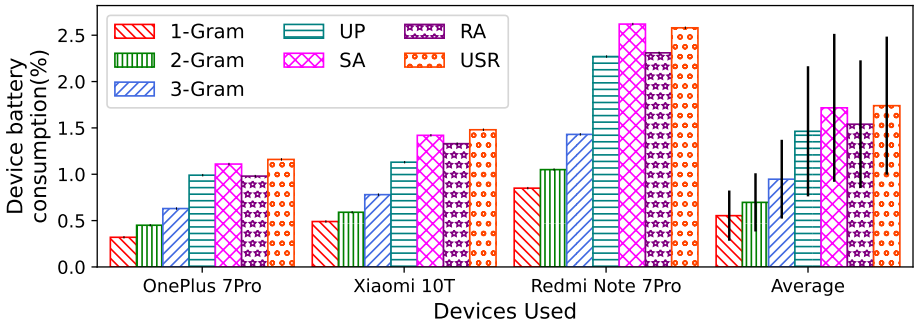
To measure the runtime efficiency (execution time and energy consumption), we have used five Android apps—Slack, UberCab, Twitter, AarogyaSetu, and Facebook. The selection of these apps is based on their size and the number of Dex files used (SingleDex or MultiDex file app, see Table 5.10). We analyse these apps on three different mobile devices—OnePlus 7Pro, Xiaomi 10T, and Redmi Note 7Pro. We execute each app ten times on each device and log the time taken to extract different features used in Section 5.5.1 and the device energy consumed by these methods (see Figure 5.6). To obtain the battery utilization, we have used `dumpsys` utility through ADB shell and analyzed using the `battery-historian` tool. Figure 5.6(a) shows the average time spend to extract features from an individual app executed on OnePlus 7Pro, which is obtained by taking the average of all the execution time (ten runs), whereas Figure 5.6(b) denotes the average time taken to analyse all apps on different devices. The energy consumption result (see Figure 5.6(c)) shows battery utilization in analyzing these apps ten times. For the OnePlus 7Pro device (Figure 5.6(a)), the result shows that the 2-Gram feature set takes ∼5.32 seconds, which is 2.23X and 2.53X faster than the RA and USR feature set, respectively. The feature extraction time depends on the Dex file size and not on the size of APK because an APK also contains other resources and files like images, native code, etc. With respect to device battery consumption (see Figure 5.6(c)), the 2-Gram approach also outperforms all the other methods that do not use opcode information and improves the device energy consumption by more than 2.1X (consumes only 0.45% of total device battery). However, the average execution time and energy consumption of all the devices (averaging the estimation of all the devices) for the 2-Gram feature set are 6.06 seconds




(a) Execution time of an App with different techniques on OnePlus 7Pro.




(b) Average execution time of different techniques.




(c) Device energy estimation.

FIGURE 5.6: Estimation of feature extraction time and device battery consumption of (i) 1-Gram, (ii) 2-Gram, (iii) 3-Gram, (iv) Used Permissions (UP), (v) Suspicious APIs (SA), (vi) Restricted APIs (RA), and (vii) USR (Combined UP, SA and RA).

and 0.7%, respectively. For the analysis of individual app execution time on other mobile devices, please refer to the Appendix A.2.4.

### 5.5.6 Discussion and Limitations

Even though DeepDetect provides a strong defense system to detect unseen malware on a real device with a malware detection rate of more than 97.5%, it also has the following

limitation:

***(i)*** Currently, it is designed to work as a third-party app for detecting Android malware on an actual device. Hence it cannot stop an app from being installed on a device. To overcome this limitation, a device vendor or AOSP project can include it in their core system, and the installation of an app should start after getting a clean chit from DeepDetect.

***(ii)*** It cannot detect packed malware where entire code is encrypted except for the unpacking logic.

***(iii)*** As DeepDetect uses information from Manifest file and Dex code (opcode) to detect malware, it cannot detect malware that include malicious behaviour exclusively in the native code.

***(iv)*** Static analysis based detection systems fail to detect malware that downloads malicious code from the external source and execute it at runtime. This is also true with DeepDetect. However, if a malware dynamically loads a code already present inside the APK and is not encrypted, DeepDetect can detect such malware efficiently as we extract opcode information from all the Dex file present inside an APK.

***(v)*** One concern with the DeepDetect is that it uses Dex code as a feature, and Android is increasingly moving toward ART to run an application. However, using ART in Android as the default runtime does not change the underlying instruction set (Dalvik instructions). Android only uses the Dex instruction to develop Android Apps. ART comes into the picture only at runtime. As DeepDetect only operates on the static feature and does not rely on the dynamic feature. Hence, using the Dalvik virtual machine or ART does not impact the performance of DeepDetect.

**Evaluation summary:** In a nutshell, DeepDetect can effectively detect malware and maintain its detection capability against the obfuscated samples with significantly low processing time when deployed on the device.

## 5.6 Related Work

With the rapid growth in Android malware, various defense systems have evolved to fight malware. Existing defense mechanisms [6, 8, 18, 55, 56, 66, 68, 71, 100, 134, 135, 136, 137, 138] use static, dynamic, or a combination of both analysis techniques to analyze apps [85]. These methods can be deployed either on market place in an offline manner or on a real device. Since dynamic analysis is costly and requires significantly higher processing power, deploying the same techniques on a real device is impractical. For example, Malton [138] is an on-device dynamic malware analysis system that monitors an app on each layer of Android OS. For Malton, the maximum slowdown for different Java operations is ∼36X while observing taint propagation. This solution is good for the analysis environment but not suitable for the end-users devices. Therefore, we restrict our discussion to the static analysis based on-device malware detection.

With the consideration of low-end mobile devices, some on-device malware detectors [6, 66, 67, 68, 136] have been proposed and works only on static features. Drebin [6] extracts features from the Dex code and Manifest file with a detection rate of 94%. IntelliAV [66] uses framework level API along with the features extracted from the Manifest file. IntelliAV shows a high prediction rate of more than 99% in the case of known samples (training sample), while the detection rate falls to ∼72% for unseen samples. However, the average analysis times on real devices for Drebin and IntelliAV are 10 seconds and 3.5 seconds, respectively. Directly comparing the execution time of DeepDetect with Drebin and IntelliAV is not a good approach because the average app size has quintupled [139] every year. Talos [67] uses only requested permissions and trains a deep learning model while achieving an accuracy of more than 93% and takes negligible time to analyze an app (in milliseconds). DeepDetect analysis time is ∼5.32 seconds, which is more than the Talos, but DeepDetect's malware detection rate is more than 97% for unseen malware, which is significantly high compared to Talos, Drebin, and IntelliAV.

Mercaldo et al. [136] have also proposed an on-device malware detector using 1-Gram opcode sequence only, where they utilize six opcodes (details can be seen in [136]). Our

experiment shows (see Section 5.5.5) that 1-Gram opcode's device energy consumption and average execution time for feature extraction are relatively less than the 2-Gram opcode sequence. However, the malware detection accuracy is good for the 2-Gram method compared to that of 1-Gram. Furthermore, utilizing only single-source information to design a malware detector is not a good approach where the single source is a Dex file. Therefore in our solution, we also include information from the Manifest file in the feature set.

Similarly, Yuan et al. [68] use API call information, permissions and intent filters as the feature set. As shown in Figure 5.4, API-based information is not good to detect malware as compared to Opcode because the API calls are the most susceptible to obfuscation attacks. However, this work is benefited from the on-device training to train a model incrementally to learn more malicious behavior. Even though on-device training is a good approach, an end-user device does not see a variety of samples in the live environment, which is a core requirement to learn different behavior.

Some other on-device malware detectors [137, 140] are also proposed using dynamic analysis or a mix of static and dynamic techniques (also known as hybrid analysis). Sinha et al. [140] insert instrumentation information into an app with the help of Dynalog [141] and then execute the modified app on an actual device. This method requires modification in an app due which the integrity of the app is compromised which can be exploited by malware to bypass dynamic analysis.

Similarly, BRIDEMAID [137] uses hybrid techniques to detect malware on-device. For the static analysis, BRIDEMAID uses the n-Gram opcode sequence as a feature set (similar to our approach), whereas they rely on a kernel-level modification (kernel module) for the collection of dynamic information (adapting the MADAM [11] solution). As the collection of dynamic information requires modification in the Android kernel, BRIDEMAID requires rooting of the device or support from the device vendor/AOSP.

## 5.7 Summary

As Android's official market place (Play Store) itself is not free from malware, we have proposed DeepDetect: an on-device malware detector that is capable of detecting malware on a real device. In DeepDetect, we have designed a feature engineering framework with the aim to reduce the feature set size by finding the right tradeoff of feature set size and detection accuracy. We have shown the effectiveness of the feature engineering framework by reducing the feature set size from 712K to 75 features that are free from the impact of code obfuscation. We have performed experiments to demonstrate the effectiveness of DeepDetect against the known, unseen, and obfuscated malware samples, and shown that DeepDetect can effectively detect more than 97% new malware with an FPR of 1.73%. For the obfuscated malware, DeepDetect achieves a malware detection rate of 95.57%. However, the malware detection rate of DeepDetect for known malware is 99.90%, with an FPR of 0.01%. Additionally, DeepDetect performance is not impacted significantly when evaluated without the spatial and temporal biases, and achieves malware detection rate of ∼97%. Finally, we have shown that when DeepDetect deployed on a real device, it can analyze an app in ∼5.32 seconds on an average, which is 2.23X faster than API based malware detector, while consuming 0.45% (for 50 apps) of total device battery.

Aside from detecting malware, classifying the malware's family allows security analysts to reuse malware removal techniques that have been proven to work for that family of malware. Family information also aids in the articulation of the damages done by malware. Therefore, identification of the malware family is also critical. In the next chapter, we introduce the automatic identification of the malware family.

# Chapter 6

# MAPFam: Android Malware Family Classification

In this past, more attention was given to malware detection rather than identification of family of a malware in which it belongs to. Classifying the family the malware belongs to, helps security analysts to reuse malware removal techniques that is known to work for that family of malware. It takes manual analysis if a malware belongs to an unknown family. Therefore, classifying malware into exact family is important. This chapter presents a technique and tool named MAPFam that applies machine learning on static features from the Manifest file and API packages to classify an Android malware into its family. This work is premised on a starting hypothesis. First, we introduce with the hypothesis and the testing process of it. Later, we design malware family identification framework MAPFam based on the testing result of our hypothesis followed by the evaluation of it.

## 6.1 Introduction

With the rapid growth in malware numbers, variations, and diversity, it is challenging to manually analyse the malware to obtain signatures and prepare defense tactics (e.g., employ antivirus techniques) against the new malware. Automated classification of malware

into different families can address the scale and dynamic nature of malware growth.

The process of auto-classification of malware into families can expedite the process of flagging the malware and keep the anti-virus defense mechanisms up to date by obtaining applicable signatures. On the other hand, if a malware can not be reliably mapped to a family, human expertise is warranted which may lead to creation of a new malware family. Existing mapping techniques use several features from an app to accurately classify malware into different families. In this chapter, *we analyze the efficacy* of mapping techniques based on different app features—API calls, permissions and API packages (see Section 6.4.1). Moreover, *we propose MAPFam*, a malware family classification solution using combination of app features for better accuracy of malware to family mapping.

In the past, many [6, 8, 9, 10, 11, 12, 13, 62, 14, 15, 16, 135, 142, 143] machine learning based Android malware *detection* systems have been developed. There are few [6, 9, 10, 12, 16, 18, 62, 144, 145, 146] that classifies an Android malware into malware families. Mostly, the machine learning based Android malware detection tools utilize either static features ([6, 8, 62, 142, 145]) or dynamic features ([10, 11, 142]). Some machine learning based malware analysis tool (like EC2 [18]) use both types of features i.e., static and dynamic. Generally, dynamic features are obtained by executing a malware sample on an emulated device. An emulator based dynamic analysis platform generally fails to capture malicious behavior if the malware is designed to detect emulators [74]. Hence, our study mainly focuses on static analysis based malware detectors, specifically in the domain of malware family classification.

Most static malware analysis tool [6, 12, 145] extracts features from either manifest file, Dex code or both. In the past, requested permissions from the manifest file and API call information from the Dex code are widely used for Android malware detection and family classification. Requested permissions provide an overview of the capabilities that an app has whereas, API call information is used to capture actual malicious behavior.

In this work, we start with a hypothesis we formulated based on our experience with Android malware detection. We experimentally test the hypothesis and based on the

evidences obtained – we established that the hypothesis holds. This leads to creation of a tool MAPFam – for mapping malware into their families.

### 6.1.1 The Hypothesis

The performance of API call information-based malware detectors or family classification models may be negatively impacted due to code obfuscation (use of java reflection) or obsolete API. Also, the use of API call information unnecessarily increases the size of the feature set while not contributing significantly towards malware family classification (see Section 6.4.1). Alternative to the API call information, system API packages can be used for malware detection and classification. An API package is a representative for a group of API calls. Whenever an API gets invoked, the corresponding API package is always referred. Furthermore, a system API package is free from the obfuscation attack, as its implementation details is not present in an APK. Therefore, using API package information in place of API calls will reduce the size of features and provides a better classification accuracy.

### 6.1.2 Testing the Hypothesis

This work comparatively analyzes the effectiveness of features like API package used, API call information, and requested permissions in classifying malware to their respective families (section 6.4.1). Subsequently, we design MAPFam, an Android malware family classification system that uses features from the *Manifest file* and *API packages used* to identify the malware family. In MAPFam, we first extract static features from the Android Manifest file and Dex code (API Packages) and encode them to use with a machine learning algorithm. After that, we pass these encoded features to a feature selection module to obtain optimal features that are most relevant to malware family identification. We evaluate MAPFam using AMD dataset [80] to show that MAPFam provides increased accuracy compared to techniques using only API calls (for restricted APIs) or requested permissions.

Overall, our contributions are as follows:

- We design MAPFam, an Android malware family classification model that uses static features from the manifest file and API packages (Section 6.3). To design a family classification model, we develop a feature extraction and encoding module to extract features from an Android app (Section 6.3.2). The API package information has not been evaluated/used heretofore, for android malware family classification, to the best of our knowledge.

- First, we show the effectiveness of the API package feature set against the requested permissions and API calls. API package feature set is ∼1.63X and ∼1.04X accurate compared to models that only use either APIs or requested permissions. We compute the model reliability of API-packages and it comes out to 95.83% (Section 6.4.1).

- Finally, we evaluate MAPFam against the known malware families with multiple classifier algorithms. MAPFam model can classify malware families with an accuracy above 97% and 97.55% of model reliability rate (Section 6.4.2). We also evaluate the effectiveness of MAPFam for the identification of individual malware families. The evaluation results show that MAPFam can *perfectly* identify 36 malware families out of 60 with an average precision rate of more than 97% (Section 6.4.3).

## 6.2 Android Malware Dataset (AMD)

We have used the real-world malware samples from AMD [80] which is the part of D1:Base-Dataset. AMD is the largest public dataset that contains 24,553 unique labelled malware distributed among 71 different families. Note that, the recent labelled malware is not publicly available. In the AMD dataset, the `airpush` malware family size is the largest with 7,843 unique malware whereas, the smallest size of the family contains only one sample (`roop` malware family). A sufficient amount of representative samples are needed to train and test a model for a machine learning algorithm. However, in AMD dataset,

some malware family does not have enough samples. To overcome this issue, we utilize the top 60 malware families of the AMD dataset (see Table 6.1), which accounts for 24,205 unique malware samples and have at least 9 or 10 unique malware.

TABLE 6.1: Distribution of malware family in the dataset.

| ID | Family | #Samples | ID | Family | #Samples |
|---|---|---|---|---|---|
| 0 | airpush | 7,843 | 30 | andup | 44 |
| 1 | dowgin | 3,384 | 31 | boxer | 44 |
| 2 | fakeinst | 2,172 | 32 | ksapp | 36 |
| 3 | mecor | 1,820 | 33 | gorpo | 32 |
| 4 | youmi | 1,300 | 34 | stealer | 25 |
| 5 | fusob | 1,270 | 35 | updtkiller | 24 |
| 6 | kuguo | 1,199 | 36 | zitmo | 24 |
| 7 | jisut | 558 | 37 | vidro | 23 |
| 8 | droidkungfu | 546 | 38 | aples | 21 |
| 9 | bankbot | 460 | 39 | fakedoc | 21 |
| 10 | rumms | 402 | 40 | fakeplayer | 21 |
| 11 | lotoor | 329 | 41 | ztorg | 20 |
| 12 | mseg | 235 | 42 | winge | 19 |
| 13 | boqx | 215 | 43 | penetho | 18 |
| 14 | minimob | 203 | 44 | cova | 17 |
| 15 | triada | 197 | 45 | mobiletx | 17 |
| 16 | kyview | 175 | 46 | fjcon | 16 |
| 17 | slembunk | 174 | 47 | kemoge | 15 |
| 18 | simplelocker | 172 | 48 | spambot | 15 |
| 19 | smskey | 165 | 49 | mmarketpay | 14 |
| 20 | gumen | 145 | 50 | svpeng | 13 |
| 21 | gingermaster | 128 | 51 | vmvol | 13 |
| 22 | leech | 109 | 52 | faketimer | 12 |
| 23 | nandrobox | 76 | 53 | steek | 12 |
| 24 | bankun | 70 | 54 | utchi | 12 |
| 25 | koler | 69 | 55 | fakeangry | 10 |
| 26 | mtk | 67 | 56 | opfake | 10 |
| 27 | golddream | 53 | 57 | spybubble | 10 |
| 28 | androrat | 46 | 58 | univert | 10 |
| 29 | erop | 46 | 59 | finspy | 9 |

## 6.3 Design

This section presents an overview of MAPFam: a **M**anifest file and **A**PI **P**ackage based Android malware **Fami**ly classification model, and elaborates the working of its core components.

### 6.3.1 An Overview

The main aim of this work is to develop a system that can label Android malware into its respective families without human expertise. Figure 6.1 shows the process of learning the Android malware family classification model. As shown in Figure 6.1, there are three main modules of the proposed work — *(i)* feature extraction & encoding, *(ii)* feature selection, and *(iii)* learning model. In the feature extraction & encoding module, static features are extracted from an Android app. The extracted features are then encoded to get feature vectors that are given to the feature selection module. The feature selection module selects the best features that directly contribute to the family classification work. Eventually, the learning model module trains the final classification model on the selected features. We describe the details of each of these modules in the following subsection.

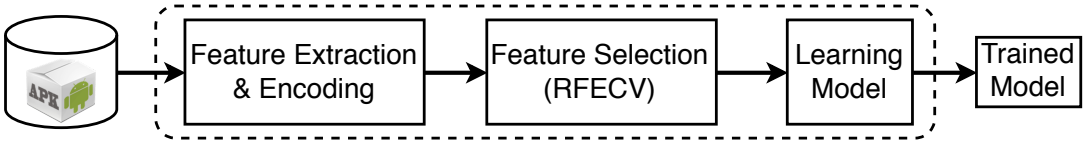



FIGURE 6.1: Architecture of learning malware family classification model.

### 6.3.2 Feature Extraction and Encoding

This module is the heart of family classification model. We extract features from two locations, i.e., *(i)* Android manifest file [19] and *(ii)* Dex code similar to Drebin [6]. However, the features extracted for this work are slightly different from Drebin [6]. In Drebin, all the features are of binary type where the value of a feature is set to one if it is present in an app otherwise zero. Whereas, in our work we use two type of features—*(i)* Numeric and *(ii)* Binary (see Table 6.2).

Furthermore, Drebin extracts used permissions, URLs, restricted APIs and sensitive APIs from the Dex code of an app, whereas we only extract the information about the API packages used by an app. The main reason of using package information is—

TABLE 6.2: Encoding scheme of static features extracted from Manifest file and Dex code.

| | Feature Type | #Features | | Source | Encoding |
|---|---|---|---|---|---|
| | | Original | Selected | | |
| Numeric (Count) | $Activities_C$ | 1 | 1 | Manifest | Number of activities |
| | $Services_C$ | 1 | 1 | Manifest | Number of services |
| | $Receivers_C$ | 1 | 1 | Manifest | Number of Receivers |
| | $Providers_C$ | 1 | 0 | Manifest | Number of Providers |
| | $Intents_C$ | 1 | 1 | Manifest | Number of intent filters |
| | $ReqPerm_C$ | 1 | 1 | Manifest | Number of requested |
| | $CstPerm_C$ | 1 | 1 | Manifest | Number of user defined permissions |
| | $Packages_C$ | 1 | 1 | Dex code | Number of system API packages used |
| Binary | $ReqPerm_B$ | 261 | 33 | Manifest | 1 if a system defined permission is declared in the Android manifest file otherwise 0 |
| | $Packages_B$ | 159 | 41 | Dex code | 1 if a system API package is used by an app otherwise 0 |
| **Total Features** | | **428** | **81** | – | – |

*(i)* Generally, a package contains multiple classes of the same type. Similarly, a class is a group of methods (APIs) and related data. Therefore, package information reduces the number of features for a malware family classification model compared to the APIs based mechanism. For example, in Android API level 30, there are 4,833 unique system-defined classes. At the same time, the count of system-defined unique packages is 226, which is 21.38X lesser than classes. Suppose a class contains an average of 4 methods (generally it is more than 4). Then there are in total of 19,332 unique APIs available as features, which is ∼85.54X larger than system package based features.

Therefore, we extract API package information from the Dex code. Features extracted from the manifest file are—

```
Activities
Services
Broadcast Receivers (receivers)
Content Providers (providers)
Intent Filters (intents)
Requested Permissions
```

The above mentioned features from manifest file and Dex code have been extracted using the *Androguard* [33] and represented as strings.

#### 6.3.2.1 Feature Encoding

The primary task of this sub-module is to encode the extracted features and generate the feature vector that can be passed to machine learning classifier for training and testing. To generate feature vector we opt following strategy:

- Initially, all features in each feature set category are binary types (represented as strings). Taking all of the feature sets as binary features will increase the size of the feature set because some of the features, such as activities, providers, and others, have user-defined names that can differ from one sample to another. By counting such features, we can combine them into a single feature. This transformation drastically reduces the size of the feature set.
- Secondly, the features that have a predefined name like API packages and system defined requested permissions are considered to be used as a binary feature.
- Lastly, we also consider the number of API packages, total requested permissions, and custom permissions count as features. A custom permission is the permission which is defined by an app. As custom permission can take any name, hence we use number of custom permission as feature instead of binary.

Using the strategy mentioned above, this module produces two types of features, i.e., numeric and binary features. In Table 6.2, column *Feature Type* describes the name of a feature. In the feature name, suffix C denotes that the feature is of Numeric type, whereas suffix B denotes binary feature. Similarly, sub-column *Original* shows the total number of unique features in the feature vector after the feature encoding step. Finally, we are left with the 428 unique features, which we pass to the feature selection module (see Section 6.3.3) to find optimal features.

### 6.3.3 Feature Selection (RFECV):

After feature encoding, the resulting features are used to identify optimal features that directly contribute to the Android malware family classification task. To determine optimal features, we used RFECV (recursive feature elimination with cross validation) [130]. RFECV uses a feature ranking method and selects the best feature that contributes more in solving the desired problem. It takes a classifier $C$ (RandomForest), a ranking function $F$ (accuracy) and the number of features $N$ (set to 1) to eliminate in each steps. As a result, RFECV provides a grid of score and the set of optimal features that gives highest accuracy. The grid of score provided by the RFECV is shown in Figure 6.2 as accuracy vs. #features graph, and the optimal number of features obtained is listed in sub-column *Selected* of Table 6.2 (total 81 unique features).

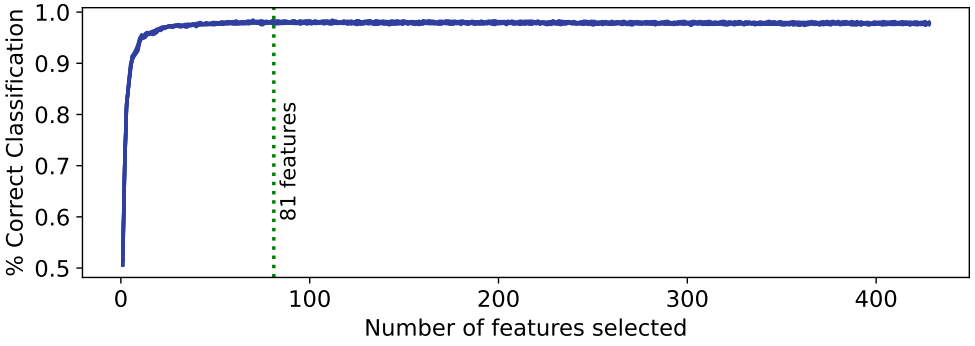



FIGURE 6.2: Feature selection using RFECV.

### 6.3.4 Learning Model

The final features obtained from Section 6.3.3 contains optimal features that directly contibutes to the malware family classification work. For modeling the final malware family classifier, we use ExtraTree classifiers. ExtraTree is an ensemble learning based classifier which internally uses multiple Decision Tree classifier. ExtraTree classifier takes the number of estimators for training a machine learning model, which we set to 18. The final model has been trained on the randomly selected 70% of samples of our dataset,

while we kept the remaining 30% samples for the evaluation. We show the evaluation of MAPFam in Section 6.4.

## 6.4 Evaluation

To evaluate the effectiveness of MAPFam in terms of labeling a malware with its appropriate family, we have separated 30% samples (referred to as evaluation set) from the dataset (see Section 6.2) and assumed them to be unknown malware family. The rest of the 70% malware sample (referred to as training set) are used for training in every experiment including the feature selection step (see Section 6.3.3). This section answers the following research question to evaluate the proposed malware family labeling (classification) system effectively:

***(i)* Effectiveness of API Packages** (Section 6.4.1): What is the effectiveness of API packages against the requested permissions and restricted APIs?

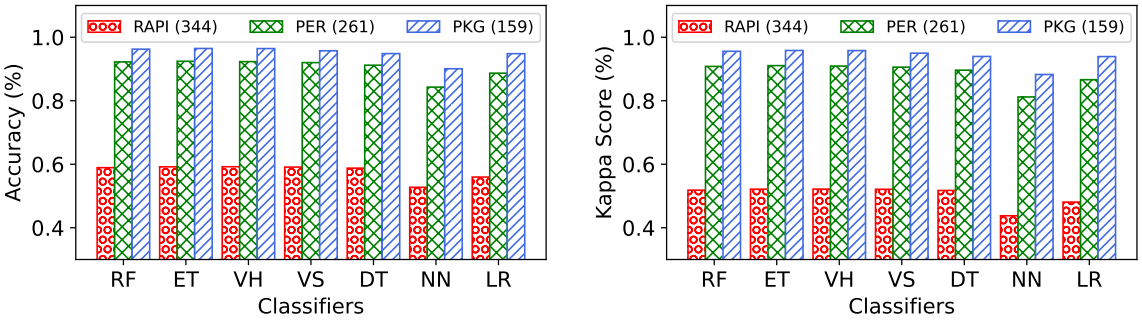
***(ii)* Performance with different classifiers against unknown malware family** (Section 6.4.2): How well MAPFam model (original and reduced feature set) performs to label unknown malware families by training a model with different classifiers?

***(iii)* Correctness of labeling of individual malware family** (Section 6.4.3): How well final model of MAPFam classifies a malware into respective family?

### 6.4.1 Performance Comparison of Features

In this experiment, we extract three categories of features from Android manifest file and Dex code—*(i)* restricted APIs (RAPI), *(ii)* requested permissions (PER), and *(iii)* API package (PKG). To compare the performance of these feature sets, we train seven different classifiers on the training set, namely *(i)* RandomForest (RF), *(ii)* ExtraTree (ET), *(iii)* Voting classifier in hard mode (VH), *(iv)* Voting classifier in soft mode, *(v)* Decision Tree,

*(vi)* Neural Network, and *(vii)* Logistic Regression, and evaluates against the evaluation set. We have used three tree-based classifiers for the voting classifier (in hard and soft voting mode), namely—RandomForest, ExtraTree, and Decision Tree. Figure 6.3 shows the performance comparison result of different features set (i.e., RAPI, PER, and PKG) with two evaluation metrics—*(i)* Accuracy and *(ii)* Cohen's Kappa score. The accuracy represents the number of samples (in percentage) correctly classified by a classifier, whereas the Kappa score is a quantitative measure of reliability for two observers for labeling the samples. In our case, the first observer is labeled dataset (AMD) which provides actual label for malware samples, and the second observer is a classifier. In other words, the Kappa score is used to measure the reliability of a machine learning model (classifier).




(a) Accuracy of family classification models.

(b) Cohen's Kappa score.

FIGURE 6.3: Accuracy and Cohen's Kappa score for the model build on different features (i) restricted APIs (RAPI), (ii) requested permission (PER), and (iii) API package (PKG).

Suppose we observe accuracy (see Figure 6.3(a)) for all the features set in every classifier, in that case, the API package features are more accurate in terms of labeling the malware family. API package-based ExtraTree classifier can correctly label 96.46% malware samples into the respective family, which is ∼1.63X and ∼1.04X accurate from restricted APIs and requested permissions based classifier, respectively. Similarly, when we measure the reliability of a model, in that case, the API package-based model is much more reliable with a 95.83% reliability rate, whereas the restricted APIs and requested permissions based classifier are 52.14% and 91.06% reliable, respectively (see Figure 6.3(b)). Therefore, API package-based feature set is more suitable as compared to APIs and permissions. Similarly,

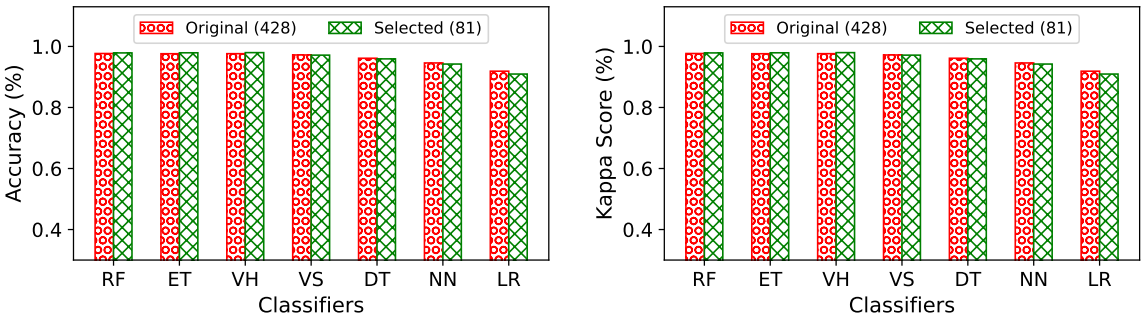
permissions are more reliable than APIs. Hence we select the requested permissions and API Package feature set for our family classification model.

**Summary:** In general, API packages based feature set is more effective and reliable as compare to API call and requested permissions.

### 6.4.2 Evaluation Against Unknown Malware Family with Different Classifiers

To show the effectiveness of MAPFam before and after feature selection to predict unknown malware family, we use evaluation set. For the training, we use the same seven classifiers as used in Section 6.4.1 and train them on the training set. Figure 6.4 shows the accuracy and Kappa score for the identification of unknown malware family by the MAPFam. When the model is evaluated without feature selection, it correctly labels more than 91% unknown malware family (see Figure 6.4(a)). The lowest accuracy achieved is 91.89% when the classifier choice is Logistic Regression, whereas the highest accuracy achieved by MAPFam is 97.62% when classifier choice is ExtraTree. However, when the feature selection is applied on the MAPFam, it correctly classifies more than 90% sample with 90.95% lowest accuracy for Logistic Regression and 97.92% highest accuracy when the classifier is ExtraTree. From the Figure 6.4(a), it is clear that MAPFam performance increases after feature selection when ExtraTree classifier is used for final model whereas performance decreases if the choice of classifier is Logistic Regression. The primary reason for this behavior is due to the unbalanced dataset and feature set size. ExtraTree can efficiently handle the unbalanced dataset with less number of features, whereas Logistic Regression performs well when the feature set size is large.

Similarly, when we measure the reliability of MAPFam (see kappa score in Figure 6.4(b)), best and worst reliable classifiers are ExtraTree and Logistic Regression, respectively. For the ExtraTree classifier, reliability rate goes up from 97.19% to 97.55% when feature selection is applied. However, in the case of Logistic Regression, it goes down from 90.45% to 89.33%.

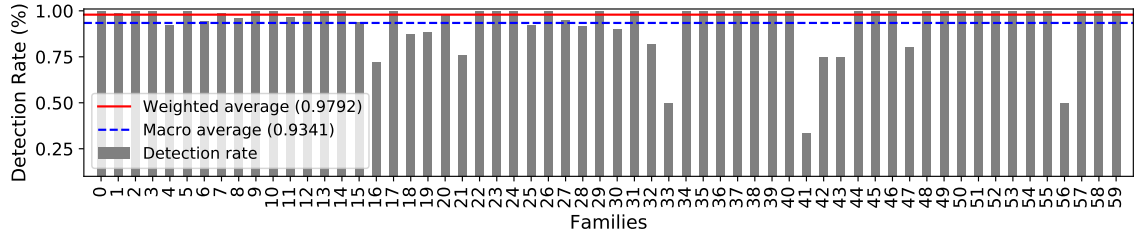



(a) Accuracy of MAPFam model.

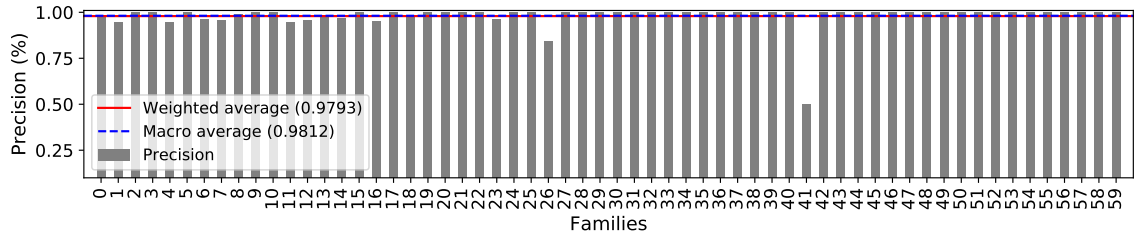
(b) Cohen's Kappa score.

FIGURE 6.4: Evaluation of final model against unknown malware family with different classifiers.

**Summary:** In general, MAPFam can correctly classify 97.92% of unknown malware into their respective families with 97.55% of reliability rate.




(a) Detection rate of individual malware family.




(b) Precision for detecting individual malware family.

FIGURE 6.5: Performance of final model for detecting individual Android malware family.

### 6.4.3 Detection of an Individual Malware Family

In this experiment, we analyze MAPFam performance for the classification of an individual malware family. To evaluate the model, we train ExtraTree classifier on the MAPFam feature set after performing feature selection (81 features), and the evaluation results are shown in Figure 6.5. Figure 6.5(a) shows the detection rate for an individual malware

family along with the macro and weighted average detection. Here, the detection rate represents the percentage of samples a model can correctly classify for a single family. Evaluation results show that MAPFam can reliably detect most families with a macro and weighted average detection rate of 93.41% and 97.92%, respectively. There are only three malware families for which the detection rate is ≤50%, and these malware families are `gorpo` (33 with 50% detection), `ztorg` (41 with 33.33% detection), and `opfake` (56 with 50% detection). Otherwise, all other malware families are reliably detected with more than 75% of detection rate. Out of 60 malware families, 36 are perfectly labeled by the MAPFam with 100% detection rate and 12 families with a detection rate of more than 90% but not 100%. Apart from the detection rate, we also measure the precision for each malware family shown in Figure 6.5(b). Results show that MAPFam is able to detect an individual malware family precisely with macro and weighted average precision of 98.12% and 97.93%, respectively. There are only two malware families for which the precision rate is less than 85%. One is `ztorg` (41) with a precision of 50%, and the other is `mtk` (26) with 84.21% precision, rest of the malware families are precisely classified with more than 94% precision rate.

### 6.4.4 Discussion and Limitations

As MAPFam makes use of API-package information for Android malware family classification, which correctly classifies 97.92% of unknown malware families. However, we do not know its runtime performance when the same technique is used for family classification on a real smartphone. In future, we would like to analyze it and develop an on-device Android malware family classification system. Furthermore, we would like to evaluate MAPFam with latest labelled dataset whenever it is publicly available or by creating our own latest dataset. Even though MAPFam provides good classification results, it also has the following limitations:

***(i)*** It cannot identify the family of malware that is encrypted.

***(ii)*** If malware tries to load a code from an external source dynamically, MAPFam cannot predict its family. However, if code is present inside the APK, MAPFam can predict the family for such malware because we check all the Dex files bundled with an APK.

***(iii)*** If a combination of two or more apps are performing a malicious activity, and one app alone is provided to identify its family, in that case, MAPFam cannot predict its family. However, if the combination of apps involved in malicious activities is provided, then MAPFam can identify its family by constructing a combined feature vector from the apps.

## 6.5 Related Work

With the rapid growth in malware numbers, variations, and diversity, various defense system has evolved to detect and categorize malware. The existing solutions [6, 8, 9, 10, 11, 12, 13, 14, 15, 16, 62, 135, 142, 143] uses static, dynamic, or combination of both (hybrid) techniques for malware detection. The main aim of malware detection work is to predict whether a new sample is malicious or benign. However, it does not provide information about a family to which malware belongs. If a malware family is known, then the same removal techniques can be reused that has been known to work for that family of malware, and analysts can give their attention to the new samples of unknown malware family. With this goal, several efforts ([6, 9, 10, 12, 16, 18, 62, 144, 145, 146]) have been made to group similar malware into families automatically. This work is also focusing on malware family identification. Therefore, we restrict our discussion to malware family classification or characterization in the context of static and dynamic/hybrid analysis.

**Static Analysis based Classification:** Static analysis-based solutions extract information from an app without executing it. Existing solutions [6, 12, 62, 145, 147, 148] extract static information from the manifest file, Dex code, and sometimes additional information from other resources like certificate, developer information, create time, and others. Alswaina et al. [147] extract information about the permission and reduce the feature set size

by excluding the least important features (with zero importance) by using ExtraTree and achieving an accuracy of 95.97% for 28 malware families. Drebin [6] extracts more than 0.5 million binary features from the Dex code and manifest file, including permissions, API calls, URLs, components, and many more. Drebin took the top 20 malware families for the classification task and achieved an average accuracy of 93%.

Fan et al. [148] make use of frequent subgraphs to understand malware behavior with the help of a function call graph of sensitive API calls inside the Dex code. DroidLegacy [145] first partitions an APK into loosely coupled modules to identify piggybacked malware. After that, it compares the API call made by each module to the signature of each family. DroidLegacy has used 14 unique malware families for the evaluation where the size of family rise between 12 to 309 and achieves an accuracy of 94.03%.

AndMFC [12] utilizes requested permissions and API call information for classifying the Android malware family. In AndMFC, the feature importance method has been used to reduce the size of features. It selects the top 1000 important features while achieving more than 96% accuracy with a precision rate of 95.02%. XU et al. [62] extracts CFGs (control flow graph) and DFGs (data flow graph) and then generate a weighted graph by abstracting both of them. The combined weighted graph is then used to identify a malware family. They used top 20 malware families where family size ranges from 57 to 2753 unique malware samples and achieved an accuracy of 94.71%.

All the above work utilizes API call information in some form, whereas we use API package information to identify Android malware families. R-PackDroid [142] is the most closely related work that utilizes API package information to characterize and detect Mobile Ransomware. However, in later work [54], authors have again shifted their focus on API call information. Also, most of the work utilizes outdated dataset or operates on fewer families that contain a sufficient number of unique samples, whereas we evaluate our solution on a large number of families.

**Dynamic/Hybrid Solutions:** In dynamic analysis-based solutions, the behavior of an app is obtained by executing it either on an emulated platform or on an actual device.

Whereas, a hybrid solution utilizes information from both the method, i.e., static and dynamic. Several efforts [9, 10, 11, 13, 18, 144] have also been made to identify malware families using dynamic/hybrid methods. Most work [9, 11, 13, 18, 144] utilizes an emulator-driven analysis framework to capture dynamic information.

EC2 [18] uses both supervised and unsupervised machine learning techniques to predict Android malware families with the ability to identify singleton families. Wang et al. [13] use all static features except for network address and restricted API used in Drebin [6] and the dynamic information captured by executing a malware sample on CuckooDroid [38]. It utilizes the top 20 malware families containing samples until 2014 and achieves an average positive rate of 98.94% (accuracy).

Similarly, Andro-Simnet [9] also uses the hybrid method and collects dynamic logs by executing a sample on emulated device. Andro-Simnet uses a malware similarity graph (a social network analysis technique) and achieves an accuracy of 97% to classify eight malware families.

All the method provides good classification results as dynamic analysis can capture actual behavior of an app. However, these techniques use an outdated dataset that does not represent the state of toady's malware which senses the execution platform. A platform-sensing malware did not show its actual behavior on finding that an execution environment is an emulated platform [74]. Recently, a work [100] has come to hide an emulated platform from a platform-sensing malware.

## 6.6 Summary

In this chapter, we have shown that the API package-based malware family classification model is ∼1.63X more accurate than the API call-based method. Later, we have developed MAPFam, a manifest file and API package-based family classification system that is capable of predicting a malware family precisely and accurately. We have performed several experiments to show the effectiveness of MAPFam to identify unknown malware

families. The evaluation results showed that the MAPFam classification system can accurately classify malware families with an average precision and accuracy of more than 97% for the top 60 malware family. The MAPFam model is 97.55% reliable. We have also shown the effectiveness of MAPFam in identifying individual malware families and found that it can perfectly identify 36 malware families out of 60. In the next chapter, we conclude this thesis and provides directions for future research work.

# Chapter 7

# Conclusion and Future Work

This thesis has contributed to the domain of Android malware detection and classification. In this chapter, first we summarizes our contributions to this thesis (Section 7.1). Next, it suggest some future directions for extending this work (Section 7.2)

## 7.1 Conclusion

Android has become the primary target of malware writers because it is an open-source platform with the highest proportion of the worldwide smartphone market. In this thesis, we have focused on designing stealthy, reliable, and low-overhead Android malware detection and classification systems. We have studied multiple malware analysis techniques and found that they are not robust enough to address the scale and dynamic nature of malware growth. From the study of existing malware analysis techniques, we have made some observations that becomes the foundation key points for this thesis.

We created a configurable anti-emulation-detection library in Chapter 3. This library is capable of arming current or new malware with various emulation-detection capabilities to test the efficacy of well-known dynamic analysis frameworks. After that, we constructed multiple datasets to test the efficacy of malware detectors in all possible scenarios.

In Chapter 4, first, we have empirically evaluated the efficacy of existing well-known dynamic analysis frameworks against the created emulation-detection library and more than 1000 real-world malware. We discovered that existing frameworks fail to hide the underlying emulated platform. Later, we created InviSeal, a stealthy dynamic analysis framework with cross-layer profiling capabilities. We have performed various experiments to see how effective it is against all potential emulation-detection approaches. Experimental results showed that InviSeal is able to hide underline emulated platforms efficiently while incurring very low-overhead. Finally, we have demonstrated several use cases of InviSeal.

In Chapter 5, we have created DeepDetect, an on-device malware detector. DeepDetect has employed a machine learning algorithm on static features to detect malware on a real device. In this work, first, we have designed an efficient feature extraction module to extract features on a real device. Later, we developed a feature engineering framework that drastically reduces the feature set size (from 712K to 75 features) by removing irrelevant features. Finally, we have designed DeepDetect with 75 features to detect malware on a real device. DeepDetect is able to detect more than 97% of new malware with a 1.73% of the false-positive rate. DeepDetect takes ∼5.32 seconds on average to analyze an app on a real device while consuming 0.45% of total device power in analyzing 50 apps.

In Chapter 6, we have developed MAPFam, a malware family classification framework. This work was premised on a starting hypothesis that features extracted from API packages rather than on API calls lead to more precise classification. We have experimentally tested our hypothesis and showed that API package-based models provide ∼1.63X more accurate classification than an API call-based method. As a result, we have designed MAPFam, a manifest file and API package-based family classification framework that is capable of predicting a malware family precisely and accurately. We have performed several experiments to show the effectiveness of MAPFam in identifying unknown malware families. The evaluation results showed that the MAPFam classification system can accurately classify malware families with an average precision and accuracy of more than 97% for the top 60 malware families and 97.55% reliable.

## 7.2 Future Directions

In this work, we have focused on protecting app stores and end-user devices being infected by the malware by designing a stealthy dynamic analysis framework and on-device malware detector followed by identifying malware families. This section offers some future work directions for extending the work presented in this thesis.

- **Revisiting anti-emulation-detection capabilities:** Even though InviSeal provides a strong defense against all the malware samples, it falls short if an app tries to detect the Xposed framework. For example, the Snapchat app uses native code to detect Xposed [123]. It is possible because Xposed capability is limited to the framework level API only, and here detection is performed through the native code. Hence, we can develop file and system properties related anti-emulation-detection mechanism at the kernel level instead of the application layer. Moreover, the timing channel attacks like studying the graphics subsystem and finding the difference between emulated vs. real devices, can also be the source of the emulation-detection. Hence, we can also develop a strategy to fool timing channel-based emulation-detection attacks.

- **Family classification with dynamic weights:** MAPFam family classification framework uses first precision to identify the family of malware. First precision refers to the strategy of assigning the family name to malware with the one which has the highest probability. For example, if we have ten families, F1 to F10, F5 gets the highest probability for a malware sample. According to the first precision, we directly assign the family name of the malware as F5, which might not be the true family of that malware. Therefore, we can opt for higher precision, say 3rd precision or 5th precision. In that case, we will assign a family name from the top 3rd or 5th probability which the weights will decide. However, we need the weights that will guide the actual family name from the higher precision families, and we do not have the same. Hence, we can develop a strategy that dynamically assigns the weights to predict malware families based on the higher precision.

- **Effect of MAPFam on a real device:** MAPFam is a practical Android malware family classification framework that correctly classifies 97.92% of unknown malware families and is 97.55% reliable. However, we do not know its runtime performance when the same technique is used for family classification on a real smartphone. Therefore, a study can also be performed to identify the runtime efficiency of MAPFam on a real smartphone. If it is efficient and does not impact smartphone battery significantly, develop an on-device Android malware family classification system to identify family on a real device.

- **Integrate on-device solution with offline detection:** Right now, DeepDetect, InviSeal, and MAPFam are working independently. We can integrate all the solutions to perform more fine-grained malware detection and classification. For example, if DeepDetect analyzes an app on a real device, it fails to detect the malware. In that case, it can submit the same app to InviSeal for dynamic analysis. Hence, a two-stage analysis process will increase the malware detection rate.

# Appendix A

# Additional Information of DeepDetect

## A.1 Features Used in DeepDetect

The feature used in DeepDetect are as follows:

*(i)* **Numeric Features:** List of numeric feature used are as follows:

```
Activities
Services
Brodcast Receivers
Content Providers
Custom Permissions.
```

*(ii)* **Requested Permissions:** The list of requested permission used are as follows:

```
android.permission.RECEIVE_BOOT_COMPLETED
android.permission.ACCESS_NETWORK_STATE
android.permission.RECEIVE_SMS
android.permission.VIBRATE
android.permission.ACCESS_COARSE_LOCATION
android.permission.INSTALL_PACKAGES
android.permission.READ_SMS
android.permission.PROCESS_OUTGOING_CALLS
android.permission.READ_PHONE_STATE
android.permission.WRITE_EXTERNAL_STORAGE
android.permission.WAKE_LOCK
android.permission.SEND_SMS
android.permission.INTERNET
android.permission.GET_ACCOUNTS
```

```
android.permission.DISABLE_KEYGUARD
android.permission.CHANGE_WIFI_STATE
android.permission.ACCESS_WIFI_STATE
android.permission.READ_LOGS
android.permission.CALL_PHONE
android.permission.READ_CONTACTS
android.permission.GET_TASKS
android.permission.WRITE_SMS
android.permission.SET_WALLPAPER
android.permission.ACCESS_GPS
android.permission.WRITE_SETTINGS
android.permission.READ_EXTERNAL_STORAGE
android.permission.SYSTEM_ALERT_WINDOW
android.permission.RESTART_PACKAGES
android.permission.MOUNT_UNMOUNT_FILESYSTEMS
android.permission.CHANGE_NETWORK_STATE
android.permission.WRITE_CONTACTS
android.permission.CAMERA
android.permission.KILL_BACKGROUND_PROCESSES
android.permission.ACCESS_LOCATION_EXTRA_COMMANDS
android.permission.BLUETOOTH_ADMIN
android.permission.USE_CREDENTIALS
android.permission.RECORD_AUDIO
android.permission.MODIFY_AUDIO_SETTINGS
android.permission.BROADCAST_STICKY
android.permission.CHANGE_CONFIGURATION
```

***(iii)* 2-Gram Opcode Sequence:** The 2-Gram opcode sequence used in DeepDetect are shown bellow. To know about the Reduce instruction set symbols, please see Table 5.1 in Chapter 5.

```
DX,XR,VD,PG,IL,
LL,SC,SR,BV,TM,
AA,FM,OM,TR,XP,
GM,XI,PA,AT,OL,
DM,DP,FP,SX,OP,
FT,LV,MM,TP,SB
```

## A.2 Additional Experiments and Results

This section contains additional results and experiments that have been carried out to evaluate DeepDetect but did not show in Chapter 5.

### A.2.1 Performance Comparison of Features

In Chapter 5, we have only shown the ROC curve and the AUC value. Table A.1 shows the evaluation results against other metrics.

TABLE A.1: Evaluation of various feature extracted from Dex file. UP: Used Permission, SA: Sensitive APIs, RA: Restricted APIs and USR: Combined features (UP, SA and RA).

| **Features** | **#Features** | **Pre (%)** | **Rec (%)** | **F1 (%)** | **FPR (%)** |
|---|---|---|---|---|---|
| 1-Gram | 18 | 94.71 | 91.74 | 93.42 | 3.33 |
| 2-Gram | 30 | 96.26 | 94.25 | 95.39 | 2.40 |
| 3-Gram | 44 | 97.44 | 95.47 | 96.75 | 1.35 |
| UP | 59 | 92.35 | 86.59 | 90.05 | 4.10 |
| SA | 42 | 86.13 | 83.60 | 83.79 | 11.44 |
| RA | 959 | 84.23 | 88.76 | 82.72 | 18.49 |
| USR | 1060 | 96.48 | 92.10 | 95.13 | 1.10 |

### A.2.2 Performance Against Known, Unseen, and New Samples

Figure A.1 shows the performance results against the ROC curve and AUC value against known, unseen, and new Samples.


FIGURE A.1: AUC-ROC curve for the final model evaluated against the known, unseen and new samples.

### A.2.3 Performance of Restricted APIs and 2-Gram Opcode Sequence with Multiple Classifier

In this section, we compare the performance of the 2-Gram opcode sequence (2-Opc) based model with restricted APIs. Restricted APIs (or API calls) are widely used in both types of malware detection systems—*(i)* on-device (Drebin [6], IntelliAV [66]), and *(ii)* on a server in an off-line fashion (DroidSeive [7], Garcia et al. [83]). Restricted APIs are those API that requires some permission, but the same is not present in the manifest file. Therefore,

we can say that restricted APIs generally showcase the malicious intents of an app, which is the main reason for showcasing the performance of the 2-Gram opcode sequence against the restricted APIs.

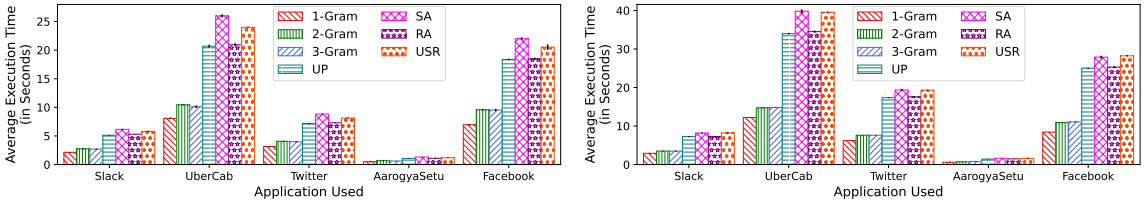
In this evaluation, we have used six classifiers to measure the performance of individual feature sets. Training and testing of the classifiers are performed on the training and evaluation set, respectively. The six classifiers that we have used are –*(i)* Random Forest, *(ii)* Extra Tree, *(iii)* Decision Tree, *(iv)* Logistic Regression, *(v)* Neural Network, and *(vi)* Nearest Neighbour. The evaluation results (see Table A.2) show that the 2-Gram opcode sequence feature set outperforms the restricted APIs in all tree-based classifiers and Neural Network classifiers with more than 85% malware detection. However, the detection rate of the 2-Gram opcode sequence is lower than Restricted APIs for Logistic regression and Nearest Neighbour. If we observe the false positive rate of these two classifiers, it is relatively very high for restricted APIs compared to the 2-Gram opcode sequence. The possible reason is that the API calls are the most susceptible to obfuscation attacks to bypass the static analysis process. Also, some APIs may go outdated or suppressed in the newer version of Android OS, which will not be present in the future/unknown apps. Hence, a model's performance may degrade when utilizing the API call information to detect malware. That is the main reason behind using opcode information in designing the on-device malware detector instead of API call information.

TABLE A.2: Comparison of 2-Gram opcode sequence (2-Opc) with restricted APIs

| **Classifiers** | **Restricted API (959)** | | | | **2-Opc (30)** | | | |
|---|---|---|---|---|---|---|---|---|
| | **Pre (%)** | **Rec (%)** | **F1 (%)** | **FPR (%)** | **Pre (%)** | **Rec (%)** | **F1 (%)** | **FPR (%)** |
| Random Forest | 85.36 | 88.76 | 84.64 | 18.48 | 96.21 | 94.25 | 96.20 | 2.4 |
| Extra Tree | 85.38 | 88.62 | 84.69 | 18.29 | 96.06 | 94.05 | 96.05 | 2.51 |
| Decision Tree | 84.82 | 88.43 | 84.02 | 19.32 | 93.28 | 93.00 | 93.25 | 6.59 |
| Logistic Regression | 81.15 | 84.30 | 80.29 | 22.82 | 73.89 | 55.85 | 72.92 | 13.49 |
| Neural Network | 81.70 | 83.28 | 81.18 | 20.51 | 87.82 | 85.04 | 87.84 | 10.13 |
| Nearest Neighbour | 83.14 | 98.06 | 74.56 | 41.55 | 91.08 | 90.90 | 91.03 | 8.91 |

### A.2.4 Run-time Efficiency

Figure A.2 shows the execution time of an app with different techniques on Xiaomi 10T and Redmi Note 7Pro.




(a) Execution time on Xiaomi 10T.

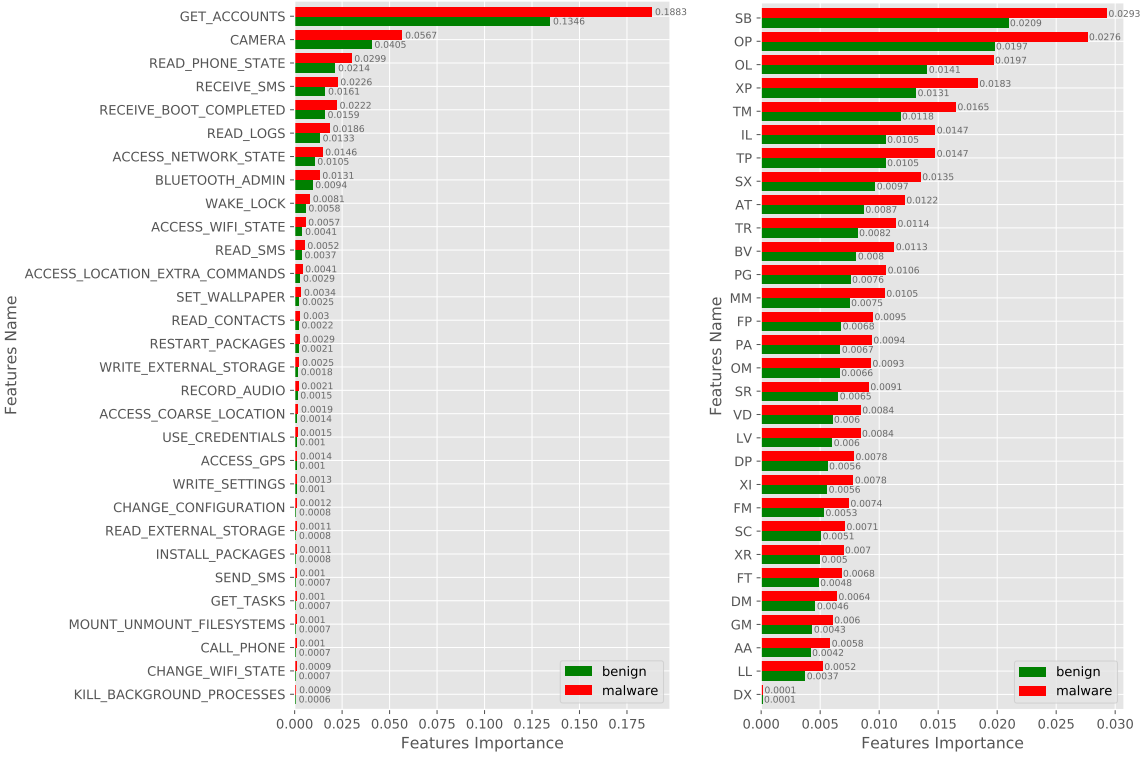
(b) Execution time on Redmi Note 7Pro.

FIGURE A.2: Execution time of an app with different techniques (i) 1-Gram, (ii) 2-Gram, (iii) 3-Gram, (iv) Used Permissions (UP), (v) Suspicious APIs (SA), (vi) Restricted APIs, and (vii) USR (Combined UP, SA and RA).

### A.2.5 Feature Importance

In this experiment, we show the quality of our features concerning classes, i.e., malware and benign. For the experiment, we train a random forest model on our training set for both categories of features individually, i.e., selected requested permissions and 2-Gram opcode sequence. Figure A.3 shows the importance of both the feature set. Figure A.3(a) shows the importance of the top 30 requested permissions, while the feature importance of the 2-Gram opcode sequence is projected in Figure A.3(b).

In Figure A.3(a), we observe a significant difference in the feature importance of GET_ACCOUNT permission between malware and benign, which indicates how well this feature can distinguish malware from the benign app. Similarly, when we observe the feature importance of the 2-Gram opcode sequence, then a Bit-wise instruction followed by a branch instruction (SB feature in Figure A.3(b)) is more important for identifying malware and benign. We observe such differences in all the features for both the feature set, which shows the quality of features in distinguishing malware from benign.

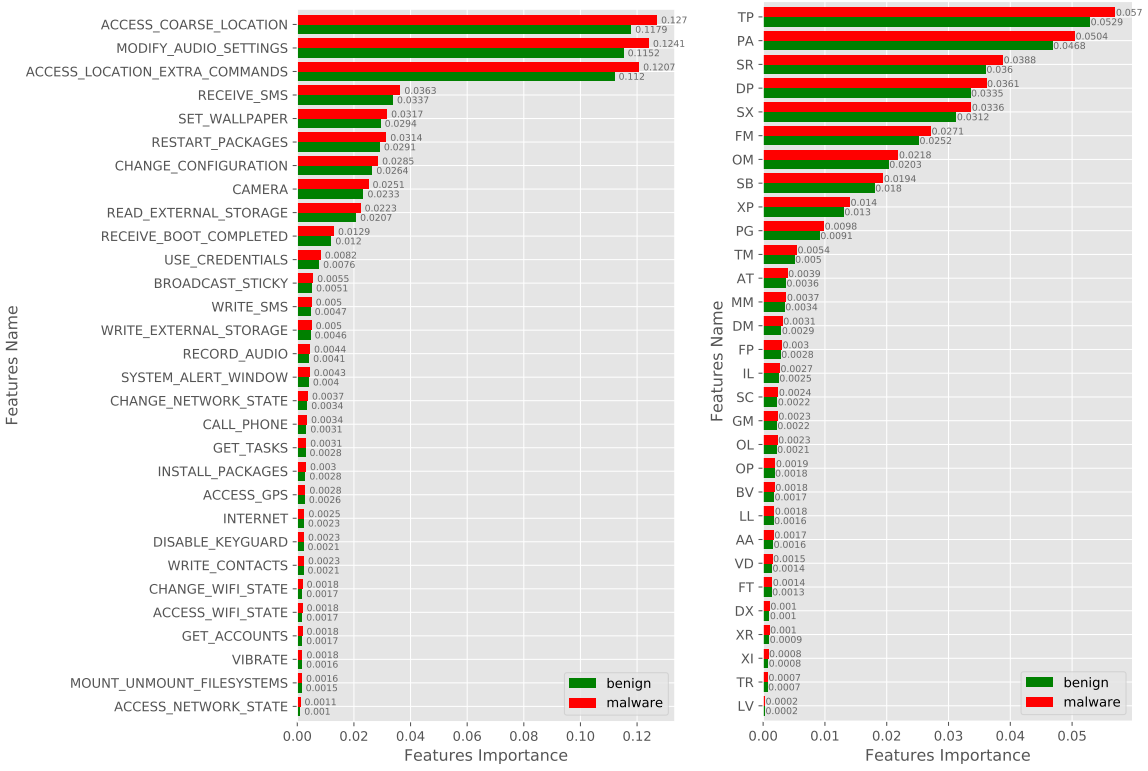



(a) Requested Permissions (top 30).

(b) 2-Gram Opcode sequence.

FIGURE A.3: Importance of features for malware detection. Figure A.3(a) shows importance of top 30 requested permissions while Figure A.3(b) shows importance of 2-Gram opcode sequence.

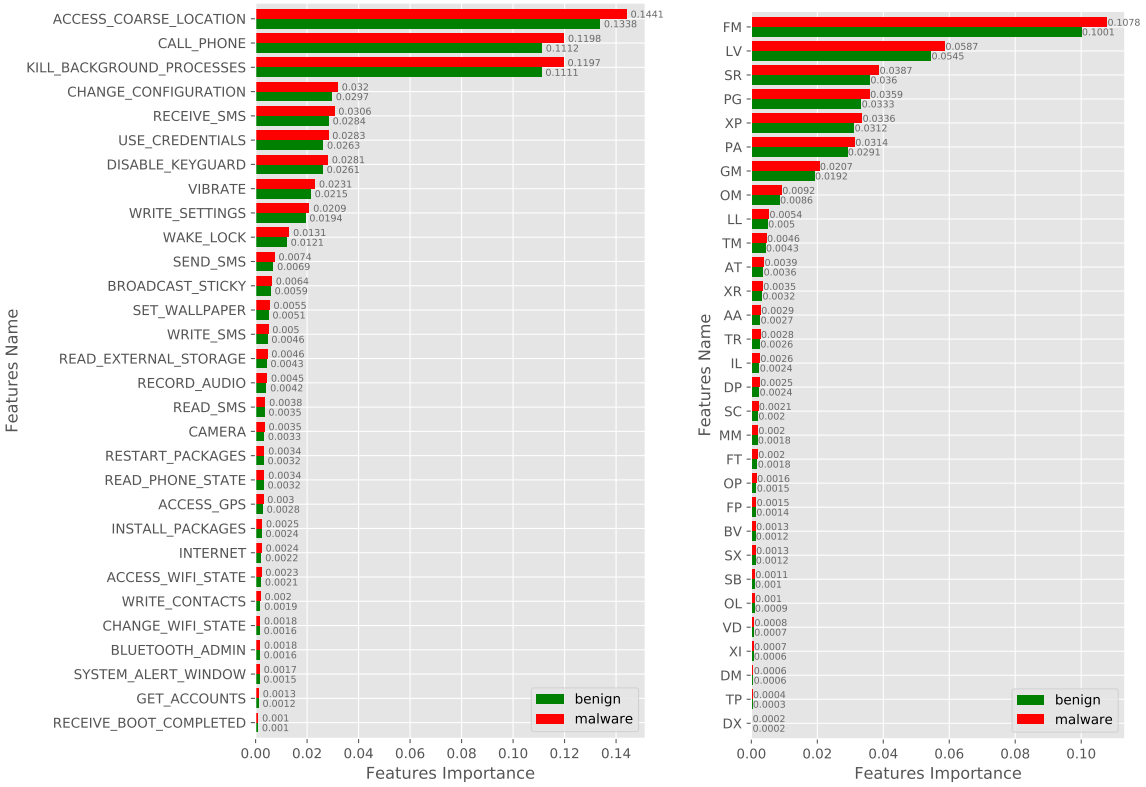
We conducted a similar experiment with new samples (AndroZoo-2019) and obfuscated malware samples by including benign samples from AndroZoo-2019. The feature importance result for new and obfuscated samples are shown in Figure A.4 and Figure A.5, respectively.




(a) Requested Permissions (top 30).

(b) 2-Gram Opcode sequence.

FIGURE A.4: Importance of features for malware detection with new sample (AndroZoo-2019). Figure A.4(a) shows importance of top 30 requested permissions while Figure A.4(b) shows importance of 2-Gram opcode sequence.




(a) Requested Permissions (top 30).

(b) 2-Gram Opcode sequence.

FIGURE A.5: Importance of features for malware detection with obfuscated samples. Figure A.5(a) shows importance of top 30 requested permissions while Figure A.5(b) shows importance of 2-Gram opcode sequence.

# Appendix B

# Additional Information of MAPFam

## B.1 Features Used in MAPFam

The features used in the final MAPFam model are as follows:

*(i)* **Numeric Features:** List of numeric features used are as follows:

```
Activities
Services
Brodcast Receivers
Content Providers
Custom Permissions
Requested Permission Count
Unique Packages Used
```

*(ii)* **Requested Permissions:** The list of requested permissions used are as follows:

```
android.permission.READ_SMS
android.permission.READ_EXTERNAL_STORAGE
android.permission.RECEIVE_SMS
android.permission.WRITE_SMS
android.permission.CHANGE_WIFI_STATE
android.permission.READ_PHONE_STATE
android.permission.WRITE_EXTERNAL_STORAGE
android.permission.SEND_SMS
android.permission.MOUNT_UNMOUNT_FILESYSTEMS
android.permission.ACCESS_FINE_LOCATION
android.permission.SYSTEM_ALERT_WINDOW
android.permission.ACCESS_WIFI_STATE
```

```
android.permission.ACCESS_COARSE_LOCATION
android.permission.RECEIVE_BOOT_COMPLETED
android.permission.VIBRATE
android.permission.GET_TASKS
android.permission.READ_LOGS
android.permission.WAKE_LOCK
android.permission.RESTART_PACKAGES
android.permission.CAMERA
android.permission.WRITE_SETTINGS
android.permission.CALL_PHONE
android.permission.WRITE_CONTACTS
android.permission.READ_CONTACTS
android.permission.PROCESS_OUTGOING_CALLS
android.permission.KILL_BACKGROUND_PROCESSES
android.permission.GET_ACCOUNTS
android.permission.ACCESS_COARSE_UPDATES
android.permission.READ_CALL_LOG
android.permission.READ_PROFILE
android.permission.ACCESS_GPS
android.permission.ACCESS_LOCATION
android.permission.ACCESS_ASSISTED_GPS
```

***(iii)* API Packages:** The API Packages used in final MAPFam model are shown bellow.

```
javax.microedition.khronos.opengles
org.xmlpull.v1
java.util.zip
android.webkit
java.util.regex
java.util.concurrent
android.telephony
java.lang.ref
android.text.format
javax.crypto
org.json
android.animation
java.text
android.accounts
java.lang.reflect
dalvik.system
java.security
java.math
android.preference
java.nio.charset
android.net.wifi
android.media
android.graphics
```

```
android.telephony.cdma
java.net
org.w3c.dom
java.util.concurrent.atomic
android.telephony.gsm
android.content.pm
android.provider
android.graphics.drawable
org.apache.http.params
android.app.admin
android.database.sqlite
javax.xml.parsers
android.content.res
java.nio
android.location
android.view.accessibility
android.database
javax.net.ssl
```

# Bibliography


[1] Malware statistics & trends report — av-test, 2022. URL `https://www.av-test.org/en/statistics/malware/`. Accessed: 1 January 2022.

[2] Android Developers. Platform architecture — Android developers, Dec 2021. URL `https://developer.android.com/guide/platform/`. Accessed: 13 December 2021.

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

[4] IDC: Smartphone Market Share-OS, Oct 2021. URL `https://www.idc.com/promo/smartphone-market-share/os`. Accessed: 12 January 2022.

[5] Number of daily Android app releases worldwide — statista.com, Dec 2021. URL `https://www.statista.com/statistics/276703/android-app-releases-worldwide/`. Accessed: 15 January 2022.

[6] Daniel Arp, Michael Spreitzenbarth, Malte Hübner, Hugo Gascon, and Konrad Rieck. Drebin: Effective and explainable detection of Android malware in your pocket. In *Proceedings of the 21st Network and Distributed Systems Security Symposium*, NDSS, 02 2014. doi: 10.14722/ndss.2014.23247.

[7] Guillermo Suarez-Tangil, Santanu Kumar Dash, Mansour Ahmadi, Johannes Kinder, Giorgio Giacinto, and Lorenzo Cavallaro. DroidSieve: fast and accurate classification of obfuscated Android malware. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, page 309–320, 2017.

[8] Anam Fatima, Saurabh Kumar, and Malay Kishore Dutta. Host-server-based malware detection system for Android platforms using machine learning. In Xiao-Zhi Gao, Shailesh Tiwari, Munesh C. Trivedi, and Krishn K. Mishra, editors, *Advances in Computational Intelligence and Communication Technology*, pages 195–205, Singapore, 2021. Springer Singapore. ISBN 978-981-15-1275-9.

[9] Hye Min Kim, Hyun Min Song, Jae Woo Seo, and Huy Kang Kim. Andro-Simnet: Android malware family classification using social network analysis. In *2018 16th Annual Conference on Privacy, Security and Trust (PST)*, pages 1–8, 2018. doi: 10.1109/PST.2018.8514216.

[10] Luca Massarelli, Leonardo Aniello, Claudio Ciccotelli, Leonardo Querzoni, Daniele Ucci, and Roberto Baldoni. Android malware family classification based on resource consumption over time. In *2017 12th International Conference on Malicious and Unwanted Software (MALWARE)*, pages 31–38, 2017. doi: 10.1109/MALWARE.2017.8323954.

[11] Andrea Saracino, Daniele Sgandurra, Gianluca Dini, and Fabio Martinelli. MADAM: effective and efficient behavior-based Android malware detection and prevention. *IEEE Transactions on Dependable and Secure Computing*, 15(1):83–97, 2018. ISSN 2160-9209. doi: 10.1109/TDSC.2016.2536605.

[12] Sercan Türker and Ahmet Burak Can. AndMFC: Android malware family classification framework. In *2019 IEEE 30th International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC Workshops)*, pages 1–6, 2019. doi: 10.1109/PIMRCW.2019.8880840.

[13] Xiaolei Wang, Yuexiang Yang, and Yingzhi Zeng. Accurate mobile malware detection and classification in the cloud. *SpringerPlus*, 4, 12 2015. doi: 10.1186/s40064-015-1356-1.

[14] Ke Xu, Yingjiu Li, Robert H. Deng, and Kai Chen. DeepRefiner: multi-layer Android malware detection system applying deep neural networks. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 473–487, April 2018.

[15] Suleiman Y. Yerima, Sakir Sezer, and Igor Muttik. Android malware detection using parallel machine learning classifiers. In *2014 Eighth International Conference on Next Generation Mobile Apps, Services and Technologies*, pages 37–42, 2014. doi: 10.1109/NGMAST.2014.23.

[16] Zhenlong Yuan, Yongqiang Lu, and Yibo Xue. DroidDetector: Android malware characterization and detection using deep learning. *Tsinghua Science and Technology*, 21(1):114–123, 2016. doi: 10.1109/TST.2016.7399288.

[17] Lok Kwong Yan and Heng Yin. DroidScope: Seamlessly reconstructing the OS and dalvik semantic views for dynamic Android malware analysis. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 569–584, Bellevue, WA, August 2012. USENIX Association. ISBN 978-931971-95-9.

[18] Tanmoy Chakraborty, Fabio Pierazzi, and V. S. Subrahmanian. EC2: Ensemble clustering and classification for predicting Android malware families. *IEEE Transactions on Dependable and Secure Computing*, 17(2):262–277, mar 2020. ISSN 1545-5971. doi: 10.1109/TDSC.2017.2739145. URL `https://doi.org/10.1109/TDSC.2017.2739145`.

[19] Saurabh Kumar and Sandeep Kumar Shukla. The state of Android security. In *Cyber Security in India: Education, Research and Training*, pages 17–22. Springer Singapore, Singapore, 2020. ISBN 978-981-15-1675-7. doi: 10.1007/978-981-15-1675-7_2. URL `https://doi.org/10.1007/978-981-15-1675-7_2`.

[20] Andrew Orlowski. Google play store spews malware onto 9 million 'Droids., 2019. URL `https://www.theregister.co.uk/2019/01/09/google_play_store_malware_onto_9m_droids/`.

[21] Apache cordova, 2021. URL https://cordova.apache.org/. Accessed: 17 December 2021.

[22] Ionic - cross-platform mobile app development, 2021. URL https://ionicframework.com/. Accessed: 17 December 2021.

[23] Wikipedia. Android application package - wikipedia, Jan 2022. URL https://en.wikipedia.org/wiki/Android_application_package. Accessed 5 January 2022.

[24] Android Developers. App Manifest overview — Android developers, Jan 2022. URL https://developer.android.com/guide/topics/manifest/manifest-intro. Accessed: 17 January 2022.

[25] Android Developers. <permission> — Android developers, Jan 2022. URL https://developer.android.com/guide/topics/manifest/permission-element. Accessed: 12 January 2022.

[26] Satoshi Maruyama, Katsuhiko Tanahashi, and Takehiko Higuchi. Base transceiver station for w-cdma system. *Fujitsu Scientific & Technical Journal*, 38:167–173, 01 2002.

[27] Wikipedia. Gsm cell id, 2020. URL https://en.wikipedia.org/wiki/GSM_Cell_ID. Accessed: 21 December 2021.

[28] Wikipedia. Mobile malware - wikipedia, May 2022. URL https://en.wikipedia.org/wiki/Mobile_malware. Accessed 25 May 2022.

[29] Trend Micro. A look at google bouncer, Oct 2012. URL https://www.trendmicro.com/en_us/research/12/g/a-look-at-google-bouncer.html. Accessed: 12 April 2022.

[30] Dan Goodin. Android devices can be fatally hacked by malicious wi-fi networks — ars technica, June 2017. URL https://arstechnica.com/information-technology/2017/04/wide-range-of-android-phones-vulnerable-to-device-hijacks-over-wi-fi/. Accessed: 12 April 2022.

[31] Ashawa Moses and Sarah Morris. Analysis of mobile malware: A systematic review of evolution and infection strategies. *Journal of Information Security and Cybercrimes Research*, 4(2):103–131, Dec. 2021. doi: 10.26735/KRVI8434. URL https://journals.nauss.edu.sa/index.php/JISCR/article/view/1578.

[32] Kimberly Tam, Ali Feizollah, Nor Badrul Anuar, Rosli Salleh, and Lorenzo Cavallaro. The evolution of Android malware and Android analysis techniques. *ACM Computing Surveys*, 49(4), jan 2017. ISSN 0360-0300. doi: 10.1145/3017427. URL https://doi.org/10.1145/3017427.

[33] Sebastian Bachmann Anthony Desnos, Geoffroy Gueguen. Welcome to Androguard's documentation! — Androguard 3.4.0 documentation, Jan 2021. URL https://androguard.readthedocs.io/en/latest/. Accessed: 20 December 2021.

[34] Argus Lab. Argus SAF - argus-pag, Nov 2021. URL `http://pag.arguslab.org/argus-saf`. Accessed: 20 December 2021.

[35] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps. *ACM Transactions on Privacy and Security*, 21(3), apr 2018. ISSN 2471-2566. doi: 10.1145/3183575. URL `https://doi.org/10.1145/3183575`.

[36] Apktool - documentation, 2022. URL `https://ibotpeaches.github.io/Apktool/documentation/`. Accessed: 20 March 2022.

[37] Patrik Lantz. An Android application sandbox for dynamic analysis. Master's thesis, Lund University, Nov 2011. URL `https://www.eit.lth.se/sprapport.php?uid=595`. Accessed: 18 December 2021.

[38] Checkpoint Software Technologies. CuckooDroid book, Jan 2021. URL `https://cuckoo-droid.readthedocs.io/en/latest/`. Accessed: 18 January 2022.

[39] MobSF Team. 1. documentation · MobSF/mobile-security-framework-MobSF wiki, Mar 2020. URL `https://github.com/MobSF/Mobile-Security-Framework-MobSF/wiki/1.-Documentation`. Accessed: 12 January 2022.

[40] Mehedee Zaman, Tazrian Siddiqui, Mohammad Rakib Amin, and Md. Shohrab Hossain. Malware detection in Android by network traffic analysis. In *2015 International Conference on Networking Systems and Security (NSysS)*, pages 1–5, 2015. doi: 10.1109/NSysS.2015.7043530.

[41] Igor Jochem Sanz, Martin Andreoni Lopez, Eduardo Kugler Viegas, and Vinicius Rodrigues Sanches. A lightweight network-based Android malware detection system. In *2020 IFIP Networking Conference (Networking)*, pages 695–703, 2020.

[42] Aqil Zulkifli, Isredza Rahmi A. Hamid, Wahidah Md Shah, and Zubaile Abdullah. Android malware detection based on network traffic using decision tree algorithm. In Rozaida Ghazali, Mustafa Mat Deris, Nazri Mohd Nawi, and Jemal H. Abawajy, editors, *Recent Advances on Soft Computing and Data Mining*, pages 485–494, Cham, 2018. Springer International Publishing. ISBN 978-3-319-72550-5.

[43] Shanshan Wang, Zhenxiang Chen, Qiben Yan, Ke Ji, Lizhi Peng, Bo Yang, and Mauro Conti. Deep and broad url feature mining for Android malware detection. *Information Sciences*, 513:600–613, 2020. ISSN 0020-0255. doi: https://doi.org/10.1016/j.ins.2019.11.008. URL `https://www.sciencedirect.com/science/article/pii/S0020025519310539`.

[44] Peng Xu, Claudia Eckert, and Apostolis Zarras. hybrid-flacon: Hybrid pattern malware detection and categorization with network traffic and program code. *CoRR*, abs/2112.10035, 2021. URL `https://arxiv.org/abs/2112.10035`.

[45] Zhenxiang Chen, Qiben Yan, Hongbo Han, Shanshan Wang, Lizhi Peng, Lin Wang, and Bo Yang. Machine learning based mobile malware detection using highly imbalanced network traffic. *Information Sciences*, 433-434:346–364, 2018.

ISSN 0020-0255. doi: https://doi.org/10.1016/j.ins.2017.04.044. URL https://www.sciencedirect.com/science/article/pii/S0020025517307077.

[46] Shanshan Wang, Zhenxiang Chen, Qiben Yan, Bo Yang, Lizhi Peng, and Zhongtian Jia. A mobile malware detection method using behavior features in network traffic. *Journal of Network and Computer Applications*, 133:15–25, 2019. ISSN 1084-8045. doi: https://doi.org/10.1016/j.jnca.2018.12.014. URL https://www.sciencedirect.com/science/article/pii/S1084804518304028.

[47] Mohammad Kamel A. Abuthawabeh and Khaled W. Mahmoud. Android malware detection and categorization based on conversation-level network traffic features. In *2019 International Arab Conference on Information Technology (ACIT)*, pages 42–47, 2019. doi: 10.1109/ACIT47987.2019.8991114.

[48] Mahshid Gohari, Sattar Hashemi, and Lida Abdi. Android malware detection and classification based on network traffic using deep learning. In *2021 7th International Conference on Web Research (ICWR)*, pages 71–77, 2021. doi: 10.1109/ICWR51868.2021.9443025.

[49] José Gaviria de la Puerta, Iker Pastor-López, Borja Sanz, and Pablo G. Bringas. Network traffic analysis for Android malware detection. In Hilde Pérez García, Lidia Sánchez González, Manuel Castejón Limas, Héctor Quintián Pardo, and Emilio Corchado Rodríguez, editors, *Hybrid Artificial Intelligent Systems*, pages 468–479, Cham, 2019. Springer International Publishing.

[50] Mohammad Reza Norouzian, Peng Xu, Claudia Eckert, and Apostolis Zarras. Hybroid: Toward Android malware detection and categorization with program code and network traffic. In *Information Security: 24th International Conference, ISC 2021, Virtual Event, November 10–12, 2021, Proceedings*, page 259–278, Berlin, Heidelberg, 2021. Springer-Verlag. ISBN 978-3-030-91355-7. doi: 10.1007/978-3-030-91356-4_14. URL https://doi.org/10.1007/978-3-030-91356-4_14.

[51] Anshul Arora, Shree Garg, and Sateesh K. Peddoju. Malware detection using network traffic analysis in Android based mobile devices. In *2014 Eighth International Conference on Next Generation Mobile Apps, Services and Technologies*, pages 66–71, 2014. doi: 10.1109/NGMAST.2014.57.

[52] Charles Lever, Manos Antonakakis, Bradley Reaves, Patrick Traynor, and Wenke Lee. The core of the matter: Analyzing malicious traffic in cellular carriers. In *Proceedings of the 20th Network and Distributed Systems Security Symposium*, NDSS, April 2013.

[53] Tieming Chen, Qingyu Mao, Yimin Yang, Mingqi Lv, and Jianming Zhu. TinyDroid: a lightweight and efficient model for Android malware detection and classification. *Mobile Information Systems*, 2018:1–9, 2018.

[54] Michele Scalas, Davide Maiorca, Francesco Mercaldo, Corrado Aaron Visaggio, Fabio Martinelli, and Giorgio Giacinto. On the effectiveness of system API-related information for Android ransomware detection. *Computers & Security*, 86:168–182, 2019. ISSN 0167-4048. doi: https://doi.org/10.1016/j.cose.2019.06.004. URL https://www.sciencedirect.com/science/article/pii/S0167404819301178.

[55] Yousra Aafer, Wenliang Du, and Heng Yin. DroidAPIMiner: Mining api-level features for robust malware detection in Android. In *Security and Privacy in Communication Networks*, pages 86–103. Springer International Publishing, 2013.

[56] Xiaoqing Wang, Junfeng Wang, and Xiaolan Zhu. A static Android malware detection based on actual used permissions combination and api calls. *International Journal of Computer, Electrical, Automation, Control and Information Engineering*, 10(9):1652–1659, 2016.

[57] M. Qiao, A. H. Sung, and Q. Liu. Merging permission and api features for Android malware detection. In *2016 5th IIAI International Congress on Advanced Applied Informatics (IIAI-AAI)*, pages 566–571, July 2016. doi: 10.1109/IIAI-AAI.2016.237.

[58] Gerardo Canfora, Eric Medvet, Francesco Mercaldo, and Corrado Aaron Visaggio. Detecting Android malware using sequences of system calls. In *Proceedings of the 3rd International Workshop on Software Development Lifecycle for Mobile*, DeMobile 2015, page 13–20, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450338158. doi: 10.1145/2804345.2804349. URL `https://doi.org/10.1145/2804345.2804349`.

[59] Roopak Surendran, Tony Thomas, and Sabu Emmanuel. On existence of common malicious system call codes in Android malware families. *IEEE Transactions on Reliability*, 70(1):248–260, 2021. doi: 10.1109/TR.2020.2982537.

[60] Chen Da, Zhang Hongmei, and Zhang Xiangli. Detection of Android malware security on system calls. In *2016 IEEE Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC)*, pages 974–978, 2016. doi: 10.1109/IMCEC.2016.7867355.

[61] Mayank Jaiswal, Yasir Malik, and Fehmi Jaafar. Android gaming malware detection using system call analysis. In *2018 6th International Symposium on Digital Forensic and Security (ISDFS)*, pages 1–5, 2018. doi: 10.1109/ISDFS.2018.8355360.

[62] Zhiwu XU, Kerong Ren, and Fu Song. Android malware family classification and characterization using CFG and DFG. In *2019 International Symposium on Theoretical Aspects of Software Engineering (TASE)*, pages 49–56, 2019. doi: 10.1109/TASE.2019.00-20.

[63] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, page 259–269, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450327848. doi: 10.1145/2594291.2594299. URL `https://doi.org/10.1145/2594291.2594299`.

[64] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems*, 32(2), jun 2014. ISSN 0734-2071. doi: 10.1145/2619091. URL `https://doi.org/10.1145/2619091`.

[65] Mingshen Sun, Tao Wei, and John C.S. Lui. TaintART: A practical multi-level information-flow tracking system for Android runtime. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 331–342, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450341394. doi: 10.1145/2976749.2978343. URL `https://doi.org/10.1145/2976749.2978343`.

[66] Mansour Ahmadi, Angelo Sotgiu, and Giorgio Giacinto. IntelliAV: toward the feasibility of building intelligent anti-malware on Android devices. In *Machine Learning and Knowledge Extraction*, pages 137–154. Springer International Publishing, 2017. ISBN 978-3-319-66808-6.

[67] H. C. Takawale and A. Thakur. Talos App: on-device machine learning using tensorflow to detect Android malware. In *2018 Fifth International Conference on Internet of Things: Systems, Management and Security*, pages 250–255, 2018. doi: 10.1109/IoTSMS.2018.8554572.

[68] W. Yuan, Y. Jiang, H. Li, and M. Cai. A lightweight on-device detection method for Android malware. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, pages 1–12, 2019. doi: 10.1109/TSMC.2019.2958382.

[69] Wei-Ling Chang, Hung-Min Sun, and Wei Wu. An Android behavior-based malware detection method using machine learning. In *2016 IEEE International Conference on Signal Processing, Communications and Computing (ICSPCC)*, pages 1–4, Aug 2016. doi: 10.1109/ICSPCC.2016.7753624.

[70] Pengbin Feng, Jianfeng Ma, Cong Sun, Xinpeng Xu, and Yuwan Ma. A novel dynamic Android malware detection system with ensemble learning. *IEEE Access*, 6:30996–31011, 2018. ISSN 2169-3536. doi: 10.1109/ACCESS.2018.2844349.

[71] Lucky Onwuzurike, Enrico Mariconti, Panagiotis Andriotis, Emiliano De Cristofaro, Gordon Ross, and Gianluca Stringhini. MaMaDroid: detecting Android malware by building markov chains of behavioral models (extended version). *ACM Transactions on Privacy and Security*, 22(2), April 2019.

[72] Feargus Pendlebury, Fabio Pierazzi, Roberto Jordaney, Johannes Kinder, and Lorenzo Cavallaro. TESSERACT: eliminating experimental bias in malware classification across space and time. In *Proceedings of the 28th USENIX Conference on Security Symposium*, SEC'19, page 729–746, 2019. ISBN 9781939133069.

[73] Wei Yan, Zheng Zhang, and Nirwan Ansari. Revealing packed malware. *IEEE Security & Privacy*, 6(5):65–69, 2008. doi: 10.1109/MSP.2008.126.

[74] Timothy Vidas and Nicolas Christin. Evading Android runtime analysis via sandbox detection. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '14, page 447–458, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450328005. doi: 10.1145/2590296.2590325. URL `https://doi.org/10.1145/2590296.2590325`.

[75] Yiming Jing, Ziming Zhao, Gail-Joon Ahn, and Hongxin Hu. Morpheus: Automatically generating heuristics to detect Android emulators. In *Proceedings of*

*the 30th Annual Computer Security Applications Conference*, ACSAC '14, page 216–225, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450330053. doi: 10.1145/2664243.2664250. URL https://doi.org/10.1145/2664243.2664250.

[76] thehackernews.com. New Android malware apps use motion sensor to evade detection, 2019. URL https://thehackernews.com/2019/01/android-malware-play-store.html.

[77] Api help (dexlib2 2.2.7 api), 2022. URL https://javadoc.io/doc/org.smali/dexlib2/2.2.7/help-doc.html. Accessed: 20 March 2022.

[78] Yajin Zhou and Xuxian Jiang. Android malware genome project, 2012. URL http://www.malgenomeproject.org/.

[79] Maiorca et al. Stealth attacks: An extended insight into the obfuscation effects on Android malware. *Computers & Security*, 51:16 – 31, 2015. ISSN 0167-4048.

[80] Fengguo Wei, Yuping Li, Sankardas Roy, Xinming Ou, and Wu Zhou. Deep ground truth analysis of current Android malware. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 252–276. Springer International Publishing, 2017. ISBN 978-3-319-60876-1.

[81] VirusShare, 2018. URL https://virusshare.com/.

[82] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. AndroZoo: collecting millions of Android apps for the research community. In *Proceedings of the 13th International Conference on Mining Software Repositories*, page 468–471, 2016. ISBN 9781450341868.

[83] Joshua Garcia, Mahmoud Hammad, and Sam Malek. Lightweight, obfuscation-resilient detection and family identification of Android malware. *ACM Transactions on Software Engineering and Methodology*, 26(3), January 2018.

[84] Sagar Jaiswal. Feature engineering & analysis towards temporally robust detection of Android malware. Master's thesis, Indian Institute of Technology, Kanpur, 2019.

[85] Alireza Sadeghi, Hamid Bagheri, Joshua Garcia, and Sam Malek. A taxonomy and qualitative comparison of program analysis techniques for security assessment of Android software. *IEEE Transactions on Software Engineering*, 43(6):492–530, 2017. doi: 10.1109/TSE.2016.2615307.

[86] Sebastian Poeplau, Yanick Fratantonio, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications. In *Proceedings of the 21st Network and Distributed Systems Security Symposium*, NDSS, San Diego, CA, February 2014.

[87] Mingshen Sun, Xiaolei Li, John C. S. Lui, Richard T. B. Ma, and Zhenkai Liang. Monet: A user-oriented behavior-based malware variants detection system for Android. *IEEE Transactions on Information Forensics and Security*, 12 (5):1103–1112, may 2017. ISSN 1556-6013. doi: 10.1109/TIFS.2016.2646641. URL https://doi.org/10.1109/TIFS.2016.2646641.

[88] Andrea Continella, Yanick Fratantonio, Martina Lindorfer, Alessandro Puccetti, Ali Zand, Christopher Kruegel, and Giovanni Vigna. Obfuscation-resilient privacy leak detection for mobile apps through differential analysis. In *Proceedings of the 24th Symposium on Network and Distributed System Security (NDSS)*, San Diego, February 2017.

[89] Martin Henze, Jan Pennekamp, David Hellmanns, Erik Mühmer, Jan Henrik Ziegeldorf, Arthur Drichel, and Klaus Wehrle. CloudAnalyzer: Uncovering the cloud usage of mobile apps. In *Proceedings of the 14th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*, MobiQuitous 2017, page 262–271, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450353687. doi: 10.1145/3144457.3144471. URL `https://doi.org/10.1145/3144457.3144471`.

[90] Lok Kwong Yan and Heng Yin. DroidScope: Seamlessly reconstructing the os and dalvik semantic views for dynamic Android malware analysis. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, page 29, USA, 2012. USENIX Association.

[91] securityfair.com. Android apps use the motion sensor to evade detection and deliver anubis malware security affairs, Jan 2019. URL `https://securityaffairs.co/wordpress/80037/malware/android-apps-motion-sensor.html`. Accessed: 13 March 2019.

[92] strace(1) - linux manual page, 2022. URL `https://man7.org/linux/man-pages/man1/strace.1.html`. Accessed: 5 January 2022.

[93] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, page 41, USA, 2005. USENIX Association.

[94] Android Developers. Run apps on the Android emulator — Android developers, 2021. URL `https://developer.android.com/studio/run/emulator`. Accessed: 21 October 2021.

[95] XDA Developers. Xposed framework hub, Aug 2018. URL `https://www.xda-developers.com/xposed-framework-hub/`. Accessed: 5 January 2022.

[96] Li Li, Daoyuan Li, Tegawendé F. Bissyandé, Jacques Klein, Yves Le Traon, David Lo, and Lorenzo Cavallaro. Understanding Android app piggybacking: A systematic study of malicious code grafting. *IEEE Transactions on Information Forensics and Security*, 12(6):1269–1284, 2017. doi: 10.1109/TIFS.2017.2656460.

[97] Michael Backes, Sven Bugiel, Oliver Schranz, Philipp Von Styp-Rekowsky, and Sebastian Weisgerber. ARTist: The Android runtime instrumentation and security toolkit. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 481–495, 2017. doi: 10.1109/EuroSP.2017.43.

[98] Valerio Costamagna and Cong Zheng. ARTDroid: A virtual-method hooking framework on Android ART runtime. In *IMPS@ESSoS*, volume 1575 of *CEUR Workshop Proceedings*, pages 20–28. CEUR-WS.org, 2016.

[99] idanr. Droidmon: Dalvik monitoring framework for cuckoodroid, 2014. URL `https://github.com/idanr1986/droidmon`. Accessed: 2 March 2022.

[100] Saurabh Kumar, Debadatta Mishra, Biswabandan Panda, and Sandeep K. Shukla. STDNeut: Neutralizing sensor, telephony system and device state information on emulated Android environments. In Stephan Krenn, Haya Shulman, and Serge Vaudenay, editors, *Cryptology and Network Security*, pages 85–106, Cham, 2020. Springer International Publishing. ISBN 978-3-030-65411-5.

[101] Github - qqshow/dendroid: Dendroid source code. contains panel and apk., Jan 2015. URL `https://github.com/qqshow/dendroid`. Accessed: 20 Oct 2021.

[102] Pendragon Software Corporation. CaffeineMark 3.0 benchmark information, May 2006. URL `http://www.benchmarkhq.ru/cm30/info.html`. Accessed: 18 January 2022.

[103] Michael Hale Ligh, Andrew Case, Jamie Levy, and Aaron Walters. *The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory*. Wiley Publishing, 1st edition, 2014. ISBN 1118825098, 9781118825099.

[104] Teller Tomer and Hayon Adi. Enhancing automated malware analysis machines with memory analysis". In *Blackhat Arsenal*, pages 1–5, 2014.

[105] Android Developers. Send emulator console commands — Android developers, Dec 2021. URL `https://developer.android.com/studio/run/emulator-console`. Accessed: 17 December 2021.

[106] AT Commands - 3GPP TS 27.007, 2020. URL `https://doc.qt.io/archives/qtextended4.4/atcommands.html`. Accessed: 10 September 2021.

[107] Wikipedia. OpenCellID, 2020. URL `https://en.wikipedia.org/wiki/OpenCellID`. Accessed: 21 December 2021.

[108] Wikipedia. Haversine formula, 2020. URL `https://en.wikipedia.org/wiki/Haversine_formula`. Accessed: 21 December 2021.

[109] 504ensicsLabs. Lime linux memory extractor, March 2021. URL `https://github.com/504ensicsLabs/LiME`. Accessed: 12 January 2022.

[110] Android Developers. Android debug bridge (adb) — Android developers, Dec 2021. URL `https://developer.android.com/studio/command-line/adb`. Accessed: 10 December 2021.

[111] Rprop/libhoudini: the default ARM translation layer for x86, extracted partly from nexus player, Oct 2017. URL `https://github.com/Rprop/libhoudini`. Accessed: 20 September 2021.

[112] Harry Gonzalez. SIM card info – apps on google play, Oct 2021. URL `https://play.google.com/store/apps/details?id=me.harrygonzalez.simcardinfo&hl=en_IN`. Accessed: 20 October 2021.

[113] The volatility foundation - open source memory forensics, Jan 2022. URL `https://www.volatilityfoundation.org/`. Accessed: 11 Jan 2022.

[114] Y. Hebbal, S. Laniepce, and J. Menaud. Virtual machine introspection: Techniques and applications. In *2015 10th International Conference on Availability, Reliability and Security*, pages 676–685, Aug 2015. doi: 10.1109/ARES.2015.43.

[115] Lei Xue, Chenxiong Qian, Hao Zhou, Xiapu Luo, Yajin Zhou, Yuru Shao, and Alvin T.S. Chan. NDroid: toward tracking information flows across multiple Android contexts. *IEEE Transactions on Information Forensics and Security*, 14(3):814–828, March 2019. ISSN 1556-6013. doi: 10.1109/TIFS.2018.2866347.

[116] Haipeng Cai. Embracing mobile app evolution via continuous ecosystem mining and characterization. In *Proceedings of the IEEE/ACM 7th International Conference on Mobile Software Engineering and Systems*, MOBILESoft '20, page 31–35, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450379595. doi: 10.1145/3387905.3388612. URL https://doi.org/10.1145/3387905.3388612.

[117] Haipeng Cai and Barbara Ryder. A longitudinal study of application structure and behaviors in android. *IEEE Transactions on Software Engineering*, 47(12):2934–2955, 2021. doi: 10.1109/TSE.2020.2975176.

[118] Haipeng Cai, Xiaoqin Fu, and Abdelwahab Hamou-Lhadj. A study of run-time behavioral evolution of benign versus malicious apps in android. *Information and Software Technology*, 122:106291, 2020. ISSN 0950-5849. doi: https://doi.org/10.1016/j.infsof.2020.106291. URL https://www.sciencedirect.com/science/article/pii/S0950584920300410.

[119] arm.com. Arm confidential compute architecture – arm®, November 2022. URL https://www.arm.com/architecture/security-features/arm-confidential-compute-architecture. Accessed: 8 November 2022.

[120] Karim O. Elish, Haipeng Cai, Daniel Barton, Danfeng Yao, and Barbara G. Ryder. Identifying mobile inter-app communication risks. *IEEE Transactions on Mobile Computing*, 19(1):90–102, 2020. doi: 10.1109/TMC.2018.2889495.

[121] Irina Mariuca Asavoae, Jorge Blasco, Thomas M. Chen, Harsha Kumara Kalutarage, Igor Muttik, Hoang Nga Nguyen, Markus Roggenbach, and Siraj Ahmed Shaikh. Towards automated android app collusion detection, 2016. URL https://arxiv.org/abs/1603.02308.

[122] Marcus Botacin, Paulo Lício De Geus, and André grégio. Who watches the watchmen: A security-focused review on current state-of-the-art techniques, tools, and methods for systems and binary analysis on modern platforms. *ACM Comput. Surv.*, 51(4), jul 2018. ISSN 0360-0300. doi: 10.1145/3199673. URL https://doi.org/10.1145/3199673.

[123] AeonLucid. Snapchat detection on Android – Aeonlucid, Jun 2019. URL https://aeonlucid.com/Snapchat-detection-on-Android/. Accessed: 21 December 2021.

[124] Ui/application exerciser monkey — android developers, Jan 2022. URL https://developer.android.com/studio/test/other-testing-tools/monkey. Accessed: 21 July 2022.

[125] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. DroidBot: A lightweight ui-guided test input generator for Android. In *Proceedings of the 39th International Conference on Software Engineering Companion*, ICSE-C '17, page 23–26. IEEE Press, 2017. ISBN 9781538615898. doi: 10.1109/ICSE-C.2017.8. URL https://doi.org/10.1109/ICSE-C.2017.8.

[126] G DATA Software AG. G DATA Mobile Malware Report 2019: new high for malicious Android apps, June 2020. URL https://www.gdatasoftware.com/news/g-data-mobile-malware-report-2019-new-high-for-malicious-android-apps. Accessed: 10 January 2021.

[127] Peter F. Brown, Peter V. deSouza, Robert L. Mercer, Vincent J. Della Pietra, and Jenifer C. Lai. Class-based n-gram models of natural language. *Computational Linguistics*, 18(4):467–479, December 1992. ISSN 0891-2017.

[128] Hang DONG, Neng qiang HE, Ge HU, Qi LI, and Miao ZHANG. Malware detection method of Android application based on simplification instructions. *The Journal of China Universities of Posts and Telecommunications*, 21:94–100, 2014.

[129] Jacob Benesty, Jingdong Chen, Yiteng Huang, and Israel Cohen. *Pearson Correlation Coefficient*, pages 1–4. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. ISBN 978-3-642-00296-0. doi: 10.1007/978-3-642-00296-0_5. URL https://doi.org/10.1007/978-3-642-00296-0_5.

[130] SKLEARN: RFECV, 2019. URL https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.RFECV.html.

[131] SKLEARN: RFE, 2019. URL https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.RFE.html.

[132] Tensorflow, 2019. URL https://www.tensorflow.org/.

[133] Thomas Colthurst, Gilbert Hendry, Zachary Nado, and Sculley D. TensorForest: scalable random forests on tensorflow. In *Machine Learning Systems Workshop at NIPS*, pages 1–9. 2016.

[134] H. Fereidooni, M. Conti, D. Yao, and A. Sperduti. ANASTASIA: Android malware detection using static analysis of applications. In *2016 8th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, pages 1–5, 2016. doi: 10.1109/NTMS.2016.7792435.

[135] J. Li, L. Sun, Q. Yan, Z. Li, W. Srisa-an, and H. Ye. Significant permission identification for machine-learning-based Android malware detection. *IEEE Transactions on Industrial Informatics*, 14(7):3216–3225, 2018. ISSN 1941-0050. doi: 10.1109/TII.2017.2789219.

[136] F. Mercaldo, C. A. Visaggio, G. Canfora, and A. Cimitile. Mobile malware detection in the real world. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion*, pages 744–746, 2016.

[137] Fabio Martinelli, Francesco Mercaldo, and Andrea Saracino. BRIDEMAID: an hybrid tool for accurate detection of Android malware. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '17, page 899–901, 2017.

[138] Lei Xue, Yajin Zhou, Ting Chen, Xiapu Luo, and Guofei Gu. Malton: Towards on-device non-invasive mobile malware analysis for art. In *Proceedings of the 26th USENIX Conference on Security Symposium*, SEC'17, page 289–306, USA, 2017. USENIX Association. ISBN 9781931971409.

[139] Shrinking APKs, growing installs. How your app's APK size impacts install. — by Sam Tolomei — Google Play Apps & Games — Medium, 2017. URL `https://medium.com/googleplaydev/shrinking-apks-growing-installs-5d3fcba23ce2`.

[140] Anukriti Sinha, Fabio Di Troia, Philip Heller, and Mark Stamp. Emulation versus instrumentation for Android malware detection. In *Digital Forensic Investigation of Internet of Things (IoT) Devices*, pages 1–20. Springer International Publishing, 2021.

[141] M. K. Alzaylaee, S. Y. Yerima, and S. Sezer. Dynalog: an automated dynamic analysis framework for characterizing Android applications. In *2016 International Conference On Cyber Security And Protection Of Digital Services (Cyber Security)*, pages 1–8, 2016. doi: 10.1109/CyberSecPODS.2016.7502337.

[142] Davide Maiorca, Francesco Mercaldo, Giorgio Giacinto, Corrado Aaron Visaggio, and Fabio Martinelli. R-PackDroid: api package-based characterization and detection of mobile ransomware. In *Proceedings of the Symposium on Applied Computing*, SAC '17, page 1718–1723, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450344869. doi: 10.1145/3019612.3019793. URL `https://doi.org/10.1145/3019612.3019793`.

[143] Kai Zhao, Dafang Zhang, Xin Su, and Wenjia Li. Fest: A feature extraction and selection tool for Android malware detection. In *2015 IEEE Symposium on Computers and Communication (ISCC)*, pages 714–720, 2015. doi: 10.1109/ISCC.2015.7405598.

[144] Santanu Kumar Dash, Guillermo Suarez-Tangil, Salahuddin Khan, Kimberly Tam, Mansour Ahmadi, Johannes Kinder, and Lorenzo Cavallaro. DroidScribe: classifying Android malware based on runtime behavior. In *2016 IEEE Security and Privacy Workshops (SPW)*, pages 252–261, 2016. doi: 10.1109/SPW.2016.25.

[145] Luke Deshotels, Vivek Notani, and Arun Lakhotia. DroidLegacy: automated familial classification of Android malware. In *Proceedings of ACM SIGPLAN on Program Protection and Reverse Engineering Workshop 2014*, PPREW'14, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450326490. doi: 10.1145/2556464.2556467. URL `https://doi.org/10.1145/2556464.2556467`.

[146] Chao Yang, Zhaoyan Xu, Guofei Gu, Vinod Yegneswaran, and Phillip Porras. DroidMiner: automated mining and characterization of fine-grained malicious behaviors

in Android applications. In Mirosław Kutyłowski and Jaideep Vaidya, editors, *Computer Security - ESORICS 2014*, pages 163–182, Cham, 2014. Springer International Publishing. ISBN 978-3-319-11203-9.

[147] Fahad Alswaina and Khaled Elleithy. Android malware permission-based multi-class classification using extremely randomized trees. *IEEE Access*, 6:76217–76227, 2018. doi: 10.1109/ACCESS.2018.2883975.

[148] Ming Fan, Jun Liu, Xiapu Luo, Kai Chen, Zhenzhou Tian, Qinghua Zheng, and Ting Liu. Android malware familial classification and representative sample selection via frequent subgraph analysis. *IEEE Transactions on Information Forensics and Security*, 13(8):1890–1905, 2018. doi: 10.1109/TIFS.2018.2806891.

[149] Le Guan, Shijie Jia, Bo Chen, Fengwei Zhang, Bo Luo, Jingqiang Lin, Peng Liu, Xinyu Xing, and Luning Xia. Supporting transparent snapshot for bare-metal malware analysis on mobile devices. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, ACSAC 2017, page 339–349, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450353458. doi: 10.1145/3134600.3134647. URL `https://doi.org/10.1145/3134600.3134647`.

[150] Miguel B. Costa, Nuno O. Duarte, Nuno Santos, and Paulo Ferreira. TrUbi: A system for dynamically constraining mobile devices within restrictive usage scenarios. In *Proceedings of the 18th ACM International Symposium on Mobile Ad Hoc Networking and Computing*, Mobihoc '17, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349123. doi: 10.1145/3084041.3084066. URL `https://doi.org/10.1145/3084041.3084066`.

[151] Wikipedia. Dalvik (software) - wikipedia, Jan 2022. URL `https://en.wikipedia.org/wiki/Dalvik_(software)`. Accessed: 15 Jan 2022.

[152] Android Open Source Project. ART and Dalvik — Android open source project, Jan 2022. URL `https://source.android.com/devices/tech/dalvik/`. Accessed: 15 Jan 2022.

[153] S Wang, H Ma, and X Wu. Permissions abuse detection for Android platform based on Droidbox. In *Proceedings of the 2014 International Conference on Network Security and Communication Engineering*, NSCE 2014, pages 87–92, 07 2015. ISBN 978-1-138-02821-0. doi: 10.1201/b18660-19.

[154] Andrea Atzeni, Fernando Díaz, Andrea Marcelli, Antonio Sánchez, Giovanni Squillero, and Alberto Tonda. Countering Android malware: A scalable semi-supervised approach for family-signature generation. *IEEE Access*, 6:59540–59556, 2018. doi: 10.1109/ACCESS.2018.2874502.

[155] Veelasha Moonsamy and Lynn Batten. Android applications: Data leaks via advertising libraries. In *2014 International Symposium on Information Theory and its Applications*, pages 314–317, 2014.

[156] android.com. Android, Mar 2019. URL `https://www.android.com/`. Accessed: 11 Jan 2022.

[157] Wikipedia. Dendroid (malware) - wikipedia, 2014. URL `https://en.wikipedia.org/wiki/Dendroid_(malware)`. Accessed: 20 December 2021.

[158] VirusTotal, 2018. URL `https://www.virustotal.com/`.

[159] Contagio. Contagio mobile - mobile malware mini dump, 2019. URL `http://contagiominidump.blogspot.com/`.

[160] 172 malicious apps with 335m+ installs found on google play, 2019. URL `https://thenextweb.com/apps/2019/10/01/google-play-android-malware-2/`. Accessed: 5 April 2022.

[161] P. Faruki, A. Bharmal, V. Laxmi, V. Ganmoor, M. S. Gaur, M. Conti, and M. Rajarajan. Android security: A survey of issues, malware penetration, and defenses. *IEEE Communications Surveys Tutorials*, 17(2):998–1022, Secondquarter 2015. ISSN 2373-745X. doi: 10.1109/COMST.2014.2386139.

[162] Parnika Bhat and Kamlesh Dutta. A survey on various threats and current state of security in Android platform. *ACM Computing Surveys*, 52(1), February 2019. ISSN 0360-0300. doi: 10.1145/3301285. URL `https://doi.org/10.1145/3301285`.

[163] Philipp Kreimel, Oliver Eigner, and Paul Tavolato. Anomaly-based detection and classification of attacks in cyber-physical systems. In *Proceedings of the 12th International Conference on Availability, Reliability and Security*, ARES '17, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450352574. doi: 10.1145/3098954.3103155. URL `https://doi.org/10.1145/3098954.3103155`.

[164] Hui-Juan Zhu, Tong-Hai Jiang, Bo Ma, Zhu-Hong You, Wei-Lei Shi, and Li Cheng. HEMD: a highly efficient random forest-based malware detection framework for Android. *Neural Computing and Applications*, 30(11):3353–3361, Dec 2018. ISSN 1433-3058. doi: 10.1007/s00521-017-2914-y. URL `https://doi.org/10.1007/s00521-017-2914-y`.

[165] S. Arshad, M. A. Shah, A. Wahid, A. Mehmood, H. Song, and H. Yu. SAMADroid: A novel 3-level hybrid malware detection model for Android operating system. *IEEE Access*, 6:4321–4339, 2018. ISSN 2169-3536. doi: 10.1109/ACCESS.2018.2792941.

[166] L. D. Coronado-De-Alba, A. Rodríguez-Mota, and P. J. E. Ambrosio. Feature selection and ensemble of classifiers for Android malware detection. In *2016 8th IEEE Latin-American Conference on Communications (LATINCOM)*, pages 1–6, Nov 2016. doi: 10.1109/LATINCOM.2016.7811605.

[167] Sebastián et al. AVclass: A tool for massive malware labeling. In *Research in Attacks, Intrusions, and Defenses*, pages 230–253, Cham, 2016. Springer International Publishing. ISBN 978-3-319-45719-2.

# Publications


[1] Saurabh Kumar and Sandeep Kumar Shukla. The State of Android Security, In: Cyber Security in India. IITK Directions, 2020 `(Book Chapter)`

[2] Saurabh Kumar, Debadatta Mishra, Biswabandan Panda, and Sandeep Kumar Shukla. STDNeut: Neutralizing Sensor, Telephony System and Device State Information on Emulated Android Environments. In 19th International Conference on Cryptology and Network Security (CANS '20), 2020

[3] Saurabh Kumar, Debadatta Mishra, Biswabandan Panda, and Sandeep Kumar Shukla. DeepDetect: A Practical On-device Android Malware Detector. In 2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS '21), 2021

[4] Saurabh Kumar, Debadatta Mishra, and Sandeep Kumar Shukla. Android Malware Family Classification: What Works – API Calls, Permissions or API Packages?. In 2021 14th International Conference on Security of Information and Networks (SIN '21), 2021

[5] Saurabh Kumar, Debadatta Mishra, Biswabandan Panda, and Sandeep Kumar Shukla. AndroOBFS: Time-tagged Obfuscated Android Malware Dataset with Family Information. In 19th International Conference on Mining Software Repositories (MSR '22), 2022

[6] Saurabh Kumar, Debadatta Mishra, Biswabandan Panda, and Sandeep Kumar Shukla. InviSeal: A Stealthy Dynamic Analysis Framework for Android Systems. ACM Digital Threats: Research and Practice `(Accepted)`


**Other Publications (not included in the thesis)**


[7] Anam Fatima, Saurabh Kumar, and Malay Kishore Dutta. Host-Server-Based Malware Detection System for Android Platforms Using Machine Learning. In Advances in Computational Intelligence and Communication Technology, 2019

[8] Arun KP, Saurabh Kumar, Debadatta Mishra, and Biswabandan Panda. SniP: An Efficient Stack Tracing Framework for Multi-threaded Programs. In 19th International Conference on Mining Software Repositories (MSR '22), 2022

[9] Vikas Maurya, Rachit Agarwal, Saurabh Kumar, and Sandeep Kumar Shukla. EPASAD: Ellipsoid Decision Boundary Based Process-Aware Stealthy Attack Detector. International Journal of Critical Infrastructure Protection `(Submitted)`