University of Brasília

Exact Sciences Institute
Computer Science Department

On the Effectiveness of the Mining Android Sandbox Approach for Malware Detection

Francisco Handrick Tomaz da Costa

Thesis submitted in partial requirement for the
Doctoral Degree in Informatics

Advisor
Prof. Dr. Rodrigo Bonifácio de Almeida

Brasília
2025

University of Brasília

Exact Sciences Institute
Computer Science Department

On the Effectiveness of the Mining Android Sandbox Approach for Malware Detection

Francisco Handrick Tomaz da Costa

Thesis submitted in partial requirement for the
Doctoral Degree in Informatics

Prof. Dr. Rodrigo Bonifácio de Almeida (Advisor)
CIC/UnB

Prof. Dr. Rodrigo Bonifácio de Almeida
Brasilia University

Prof.a Dr.a Genaína Nunes Rodrigues
University of Brasília

Prof. Dr. Eduardo Luzeiro Feitosa
Federal University of Amazonas

Prof. Dr. Breno Alexandro Ferreira de Miranda
Federal University of Pernambuco

Prof. Dr. Rodrigo Bonifácio de Almeida
Coordinator of Postgraduate Program in Informatics

Brasília, May 12, 2025

Dedicated to

I dedicate this work to everyone who has supported me throughout these years of research. To my mentors, friends, my parents, and especially my wife and daughters, for their patience and support during the most challenging times of this journey. Thank you very much!

Acknowledgements

Working towards a PhD degree is a journey which cannot be undertaken alone. Let me thank my parents for providing me with a quality education throughout my school years, which made it possible for me to get to where I am today. Without them, none of this would have been possible.

To all the students from the University of Brasília who supported me during these years of research, and with whom I had the honor of working throughout this journey. The success of this work would not have been possible without their generosity.

I would also like to thank the University of Brasília, represented by all the professors from CIC, and TU Darmstadt in Germany, especially Prof. Dr.-Ing. Mira Mezini and Dr. Krishna Narasimhan. Thank you all for believing in my potential and providing me with the human and material support necessary to develop this work.

To my advisor, Prof. Rodrigo Bonifácio, I will be eternally grateful for all the patience, attention, and energy he devoted to me. I will carry your teachings with me for the rest of my life. My deepest thanks.

To my wife, Lucianna, and my daughters, Gabriela and Alice, not only for your patience but for all the strength you gave me, especially during the most challenging times of this journey. You were incredibly important.

Finally, to the Federal District Environmental Sanitation Company (CAESB), thank you for the support and trust throughout this journey. I hope to give back to CAESB and my country by becoming a better professional and applying everything I have learned.

To everyone in general who supported me, thank you very much.

.

Do or do not. There is no try.
Master Yoda

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Análise da Eficácia da Abordagem de Mineração de Sandbox na Detecção de Malware

Resumo

Devido à popularidade da plataforma Android, aliada à relativa facilidade em aplicar técnicas de reengenharia em aplicativos Android (apps), programadores maliciosos têm se dedicado a explorar formas de ataques que visam monetizar a partir de aplicativos legítimos e violar aspectos de privacidade dos usuários. Esse cenário tem atraído a atenção de pesquisadores para o desenvolvimento de técnicas que possibilitam mitigar algumas falhas de segurança ou estratégias de ataque para aplicativos Android.

Uma iniciativa recente, proposta por Jamrozik et al., introduziu o conceito de *sandbox mining*, uma abordagem em duas fases para melhorar a segurança de aplicativos Android. Na fase de *mining* (mineração), ferramentas de geração de testes exploram o comportamento do aplicativo monitorando chamadas a APIs sensíveis. A subsequente fase de *sandbox* restringe qualquer desvio do comportamento observado durante a mineração. Esse método detecta e bloqueia chamadas não autorizadas a APIs sensíveis, melhorando assim a segurança do usuário. Posteriormente, Bao et al. estendeu o trabalho de Jamrozik et al., avaliando a eficácia da abordagem na identificação de comportamentos maliciosos e comparando as capacidades exploratórias de diferentes ferramentas de teste para *sandbox mining*. Entretanto, seu estudo apresentava limitações: não examinou completamente as contribuições das análises estática e dinâmica para o *sandbox mining*, além de suas conclusões basearam-se em um conjunto de dados limitado, com representação insuficiente de famílias de *malware*.

Nesta tese, nosso objetivo principal foi avaliar a abordagem de mineração em *sandbox* através da investigação do papel das análises estática e dinâmica na proposta. Após documentar as contribuições de ambos os métodos, realizamos um segundo estudo para verificar se a solução mantinha desempenho comparável na detecção de *malware*, quando aplicada a um conjunto de dados mais amplo e diversificado do que os utilizados em estudos anteriores. Os resultados revelaram uma queda significativa na precisão da detecção, com o F1-score diminuindo de 0,90 (em trabalhos anteriores) para 0,54 no conjunto de dados expandidos. Análises posteriores indicaram que essa degradação de desempenho foi causada principalmente por amostras de famílias específicas de *malware*, evidenciando uma limitação crítica da abordagem. Essa descoberta nos levou a investigar uma solução

complementar para abordar a vulnerabilidade identificada.

Por fim, em nosso estudo final, propusemos uma abordagem de análise de fluxo de rede aprimorada com aprendizado de máquina. Esse método demonstrou um desempenho superior na classificação de *malwares* em comparação com a mineração em *sandbox*, alcançando um F1-score de 0,85 no conjunto de dados diversificado. Notavelmente, os resultados mostraram que famílias de *malware* com baixas taxas de detecção na mineração em *sandbox* foram mais efetivamente identificadas por meio da análise de fluxo de rede, uma vez que os modelos de aprendizado de máquina, conseguiram detectar padrões característicos de atividades maliciosas.

Palavras-chave: Detecção de Malware para Android, Mineração em Sandboxes para Plataforma Android, Plataforma Android, Análise estática, Análise dinâmica e Análise de Fluxo de Rede

Abstract

Due to the widespread popularity of Android and the relative ease of reverse-engineering Android apps, malicious actors frequently exploit vulnerabilities to monetize legitimate applications and compromise user privacy. This growing threat has driven researchers to develop new techniques for mitigating security flaws and countering attack strategies targeting Android applications.

A recent initiative by Jamrozik et al. introduced *sandbox mining*, a two-phase approach to enhance Android application security. In the *mining* phase, test generation tools explore app behavior by monitoring calls to sensitive APIs. The subsequent *sandbox* phase restricts any deviations from the behavior observed during mining. This method detects and blocks unauthorized sensitive API calls, thereby improving user security. Later, Bao et al. extended Jamrozik et al.'s work by evaluating the approach's effectiveness in identifying malicious behavior and comparing the exploratory capabilities of different testing tools for *sandbox mining*. However, their study had limitations: it did not fully examine the contributions of static and dynamic analysis to *sandbox mining*, and its findings relied on a limited dataset with insufficient representation of malware families.

In this thesis, our primary objective was to evaluate the *sandbox mining* approach by analyzing the roles of static and dynamic analysis within its framework. After documenting the contributions of both methods, we conducted a second study to assess whether the solution maintained comparable malware detection performance when applied to a larger and more diverse dataset than those used in prior studies. The results revealed a significant drop in detection accuracy, with the F1-score decreasing from 0.90 (in previous work) to 0.54 on the expanded dataset. Further analysis indicated that this performance degradation was primarily caused by samples from specific malware families, highlighting a critical limitation of the approach. This finding prompted us to investigate a complementary solution to address the identified weakness.

Finally, in our final study, we proposed a machine learning (ML)-enhanced *network flow analysis* approach. This method demonstrated better malware classification performance compared to *sandbox mining*, achieving a F1-score of 0.85 in the diversified dataset. Notably, the results showed that malware families with low detection rates under *sand-*

box mining were more effectively identified through *network flow analysis*, as ML models successfully detected characteristic malicious activity patterns.

Keywords: Android Malware Detection, Mining Android Sandboxes, Android Platform, Static Analysis, Dynamic Analysis and Network Traffic Analysis

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Nowadays, the Android Operating System has dominated the market of electronic devices (including tablets and smartphones), corresponding to almost two-thirds of mobile users around the world [5, 6]. This popularity has increased the number of incidents related to malicious Android applications (malware) [1] [7, 8]. Since many tools make it easy for developers to reverse-engineer Android bytecode, decompiling and injecting malicious code into legitimate apps is straightforward. As a result, attackers can modify these apps by inserting malicious code, repackage them with harmful payloads, and redistribute them on app stores [3, 9], including official platforms like the Google Play Store [10].

Due to the malware problem and other security issues, Android security has become an important research topic, and several techniques have emerged to identify vulnerabilities in Android apps [11]. These techniques use either dynamic or static analysis to identify malicious behaviors and protect users from attacks. Dynamic analysis involves executing the program to abstract some relevant property, focusing on the actions performed dynamically [12, 13].

In contrast to dynamic analysis, static analysis aims to estimate the program behavior after scanning the source or binary code without the need to execute the program [5]. It involves the exploration of the app bytecode, for instance, mining function calls and instructions that can lead to unwanted behavior, like the leak of sensitive information, or be used to reveal the misuse of cryptographic primitives [14, 15].

Some security techniques leverage both static and dynamic analysis, though—including the Mining Android Sandbox approach (hereafter referred to as the MAS approach), which was initially designed to construct sandboxes based on calls to sensitive APIs [13]. The MAS approach works in two distinct phases. In the first phase, the MAS approach instruments an Android app to log any calls to sensitive Android methods and then executes

[1] In this thesis, we will use the terms Android Applications, Android Apps, and Apps interchangeably to represent Android software applications

the apps by leveraging test case generation tools. The idea is to explore the app's runtime behavior and collect any calls to sensitive APIs. The set of sensitive calls builds a sandbox, which is used in the second phase to block calls to sensitive APIs not observed during the initial exploratory phase. This two-step process enhances security by restricting unauthorized or unexpected access to sensitive functionalities. The MAS approach was tailored and evaluated to assess its effectiveness in detecting malicious Android apps, operating in two distinct phases [3].

1.1 Problem Statement

One effective initiative to address security issues in Android apps is isolating apps in sandboxes [13], which restricts access to sensitive data and resources. In the Android platform, for instance, developers must specify which features a particular app needs to access. For example, an app requiring access to the user's location, contact list, or camera resources must obtain explicit user authorization. If the user denies access to a specific feature, an exception will be reported if the app attempts to use that resource.

As described, the MAS approach has emerged to build fine-grained sandboxes by abstracting app behaviors based on calls to sensitive APIs. The MAS approach leverages test case generation tools to build a sandbox by monitoring the calls to sensitive Android APIs that are observed during the app execution, establishing safety rules for the sandbox [13]. The MAS approach has also been claimed as effective in detecting a popular class of Android malware based on repackaging benign apps [3], and previous studies have investigated the impact of test case generation tools on such effectiveness. For instance, Bao et al. [3] presented the results of an empirical study that compares five test case generation tools, including DroidMate [16], an automatic test generator specifically designed for use in mining sandboxes. This previous research also explored other automatic test generation tools for mining sandboxes, such as Monkey [17], GUIRipper [18], Puma [19], and Droidbot [20]. Le et al. [21] also proposed an approach to create more effective sandboxes by considering not only calls to sensitive APIs but also the actual arguments passed to the calls to sensitive APIs.

Problem 1: Jamrozik et al. argue that dynamic analysis outperforms static analysis in the context of mining sandboxes [13], although they do not provide empirical evidence to support this claim. Additionally, the MAS approach implementation by Bao et al. [3] leverages DroidFax, a static analysis tool used to instrument the Android apps in their study. However, DroidFax was originally designed for security purposes and is capable of identifying calls to sensitive APIs on its own. The specific contributions of static and dynamic analysis in the Bao et al.'s study remain unclear,

and there is a gap in the literature regarding the potential benefits of combining static and dynamic analysis in the context of MAS approach for malware detection.

Problem 2: Previous empirical studies [3, 9] suggest that the MAS approach classifies up to 70% of samples as malware in a repository containing 102 pairs of Android apps (benign and malicious). However, a dataset of only 102 app pairs may be too limited to draw sound conclusions—compromising the external validity of these findings, as it likely does not encompass a diverse enough range of malware families.

Therefore, the primary motivation for this thesis was to address the limitations in the existing literature by investigating the benefits of combining the MAS approach with static analysis, using a well-defined benchmark for comparison and a more representative dataset than those used in previous studies. Unfortunately, after experimenting with the MAS approach on a large dataset, we found that it fails to correctly classify popular families of Android malware (e.g., Gappusin), which severely compromises its performance.

Problem 3: The MAS approach approach fails to correctly classify malware from certain families, resulting in poor performance on the comprehensive dataset of 4,076 samples used in our research. In particular, the MAS approach approach achieves a F1-score of 0.89 on the dataset used in previous studies, whereas on our larger dataset, it achieves a F1-score of only 0.54.

We then explored a new approach, DroidXPFlow, which collects network traffic data by executing Android apps using test case generation tools. From this collected data, we experimented with various machine learning algorithms to classify the samples in our larger dataset as either benign or malicious. Using DroidXPFlow, we achieved a F1-score of 0.85.

1.2 Thesis Proposal

This thesis aims to conduct a comprehensive evaluation of the effectiveness of the MAS approach for malware classification. It examines the respective roles of static and dynamic analysis in this process, and proposes an enhanced methodology to address its existing limitations. To achieve these objectives, we present three supporting studies, as outlined in Figure 1.1.

In the first, we combined static and dynamic analysis algorithms to understand their benefits and limitations within the MAS approach, using the same dataset as in previous studies [3, 13]. Our findings show that integrating both techniques can effectively complement each other, leading to improved performance in the MAS approach. In the

second study, we investigated if the MAS approach scales its performance with a larger, more diverse dataset that includes additional malware families. Our results reveal that the MAS approach demonstrates lower accuracy than previously suggested, as it fails to classify certain recurring malware families. To address this issue, in the third study, we explored an alternative approach based on dynamic analysis (named DroidXPflow) that benefits from network traffic and Machine Learning (ML) algorithms. We aimed to investigate if DroidXPflow outperforms the accuracy of the MAS approach, particularly in malware families where the original approach had shown limited effectiveness. We address the following issues in the three studies:

In the first study, we explore the question (RQ1): *What are the impact of combining static analysis (e.g., FlowDroid [22] or DroidFax [23]) and dynamic analysis (e.g., MAS approach) for Malware Classification ?*

In the second study, we explore the question (RQ2): *To what extent does the use of a larger and more diverse dataset impact the performance of the MAS approach?*

In the third study, we explore the question (RQ3): *How effective is DroidXPflow, which leverages Machine Learning algorithms on data collected from network traffic, in correctly classifying Android malware ?*

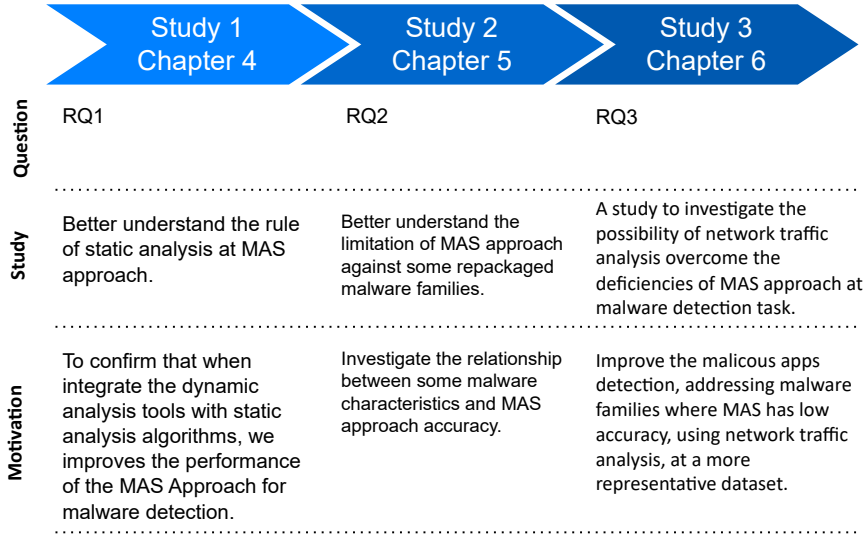


Figure 1.1: Thesis overview

1.3 Thesis Contributions

We highlight the main outcomes of this thesis, in terms of the results of the studies conducted and generated artifacts.

1.3.1 DroidXP and DroidXPflow

DroidXP is a benchmark for running experiments and comparing the performance of test case generation tools for the MAS approach. We developed DroidXP [24]—currently available in an open-source repository[2]. DroidXP is an extensible tool that allows researchers and practitioners to easily integrate new test generation tools and conduct dynamic analysis on Android apps. Its primary purpose is to explore sensitive API calls within the apps under analysis. DroidXP uses DroidFax [23] to instrument each APK file and collects data about each app's execution while a test tool (e.g., Monkey or Droidbot) is running. The development of this tool was highly valuable for our research, as we used it to reproduce previous studies on the MAS approach and to evaluate subsequent studies.

DroidXPflow represents a DroidXP extension, incorporating advanced dynamic analysis techniques to enhance malware detection capabilities. While DroidXP primarily focuses on sensitive API call analysis, DroidXPflow introduces an additional dynamic analysis component. It captures network traffic during test case execution, thereby enriching the analytical dataset with behavioral information. This enriched dataset is then processed using ML algorithms to classify apps as malware or non-malware, leveraging a pre-trained knowledge base. This integrated approach significantly strengthens the malware detection capabilities of the MAS approach by combining structural and behavioral analysis methodologies.

Figure 1.2 shows the architectural overview illustrating the integration between DroidXP and DroidXPflow. More details of both artifacts will be presented in Chapter 3 and Chapter 6.

1.3.2 On the complementarity of static and dynamic analysis

Our findings indicate that the effectiveness of DroidXP 's implementation of the MAS approach in classifying malicious applications largely stems from its integration of dynamic analysis—enabled by test generation tools—with static analysis produced through instrumentation using the DroidFax framework. Moreover, the results suggest that adopting an integrated analysis environment—combining the MAS approach with static tools such as FlowDroid and DroidFax—may yield additional benefits.

[2]https://github.com/droidxp/benchmark

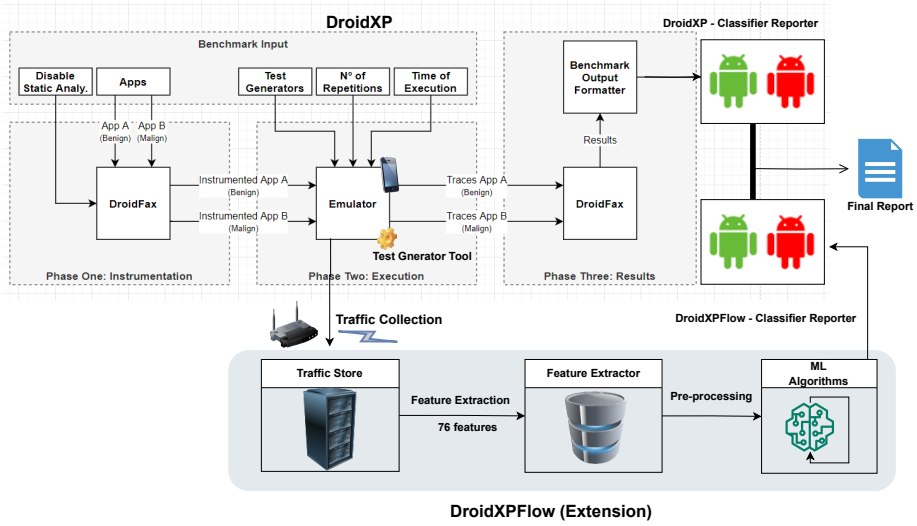


Figure 1.2: Architectural overview of the DroidXP-DroidXPflow integration

1.3.3 On the scalability of the MAS approach at a large and more diverse dataset

Our research reveals that the MAS approach performs substantially worse on a more diverse dataset compared to those used in prior work [25, 26]. Specifically, its F1-score score drops from 0.89 to 0.54. For instance, the MAS approach fails to correctly identify 89.37% of samples from the *gappusin* family and 44.44% of samples from the *revmob* family as malware. These results highlight the main reason behind the MAS approach's low recall on diverse datasets. Consequently, our findings underscore the need for supplementary techniques to enhance the MAS approach approach and improve malware detection accuracy (Section 5.3.2).

1.3.4 On the effectiveness of DroidXPflow to classify malware when compared with MAS approach

Our research shows that DroidXPflow, which integrates machine learning (ML) techniques to analyze network flow data, outperforms the MAS approach by achieving a F1-score score of 0.85 on our more diverse dataset. This result underscores its effectiveness as a promising approach for malware classification, particularly for families such as *gappusin*

and *revmob*, which exhibit frequent malicious network behavior. Notably, these are the same families for which the MAS approach demonstrated poor recall in prior studies.

1.4 Research Papers

The work associated with this thesis has resulted in the following research papers, which have already been published or are currently under review.

1. Francisco Handrick da Costa et al., "DroidXP: A Benchmark for Supporting the Research on Mining Android Sandboxes," 2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM), 2020, pp. 143-148, doi: 10.1109/SCAM51674.2020.00021.

2. Francisco Handrick da Costa, "On the Interplay Between Static and Dynamic Analysis for Mining Sandboxes," 2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), Madrid, ES, 2021, pp. 315-319, doi: 10.1109/ICSE-Companion52605.2021.00135.

3. Francisco Handrick da Costa, Ismael Medeiros, Thales Menezes, Joao Victor da Silva, Ingrid Lorraine, Rodrigo Bonifacio, Krishna Narasimhan, Marcio Ribeiro. "Exploring the use of static and dynamic analysis to improve the performance of the mining sandbox approach for android malware identification," Paper published at The Journal of Systems & Software 183 (2022) 111092.

4. Francisco Handrick Costa, Ismael Medeiros, Leandro Oliveira, Rodrigo Bonifacio, Krishna Narasimhan, Mira Mezini, Marcio Ribeiro "Scaling Up: Revisiting Mining Android Sandboxes at Scale for Malware Classification". Paper published at ECOOP 2025. Article No. 40; pp. 40:1–40:26

5. Francisco Handrick Costa, Roberto Luis Valera, Rodrigo Bonifacio, Eduardo Monteiro de Gomes, Joao Jose Gondim. "Improving Mining Android Sandbox with Network Flow Data and Machine Learning." Chapter not submitted for publication while writing this Thesis.

1.5 Thesis Organization

The remainder of this thesis is structured as follows:

- Chapter 2 presents background materials on Android, sandboxes, Android mining sandboxes, taint analysis, and symbolic execution.

- Chapter 3 describes our benchmark tool, which helped us integrate test case generation tools and compare their performance in the MAS approach.
- Chapter 4 explores the rule of static and dynamic analysis at MAS approach.
- Chapter 5 discusses a replication and extension study of MAS approach for Malware Detection.
- Chapter 6 presents how network traffic analysis and machine learning (ML) can be combined to improve malware detection.
- Chapter 7 concludes the thesis, presenting implications and future work.

Chapter 2

Background and Related Work

In this background Chapter, we present some concepts definitions, techniques, and approaches that are important to understand the remainder of this thesis document. As a first step, we introduce the Android framework, its architecture, and Android security issues at Section 2.1 and Section 2.2. Section 2.3 we introduce *Mine Sandbox* technique, and how this approach can deal with malware detection at Android apps. Since modern malware detection tools make use of static and dynamic code analysis approaches to detect unwanted behaviors, we introduce these techniques in Section 2.4 . The last sections, we list 2 code analysis techniques that we refer to in the remainder of this thesis document, taint analysis and symbolic execution (Section 2.5 and Section 2.6).

2.1 Android

Android is an open source software framework for mobile communication devices, with an advanced RISC Machine (ARM) architecture [27][28]. It incorporates an Operating System (OS), an app development toolkit assisting developers and an app framework. The Android OS is Linux-based kernel, and was developed for several devices including tablets, smartphones and others electronic devices. His open source and unrestricted app market, have made it a popular platform for third-party apps, and its primary market source for Android apps, Google Play Store, has included more apps than the app Apple Store [29] since 2011. The next subsections discuss about its architecture and security issues.

2.1.1 Android Architecture

Figure 2.1 presents the Android app stack or Android architecture. It consists of several layers like Linux kernel, native libraries, and Application Framework, for instance [1].

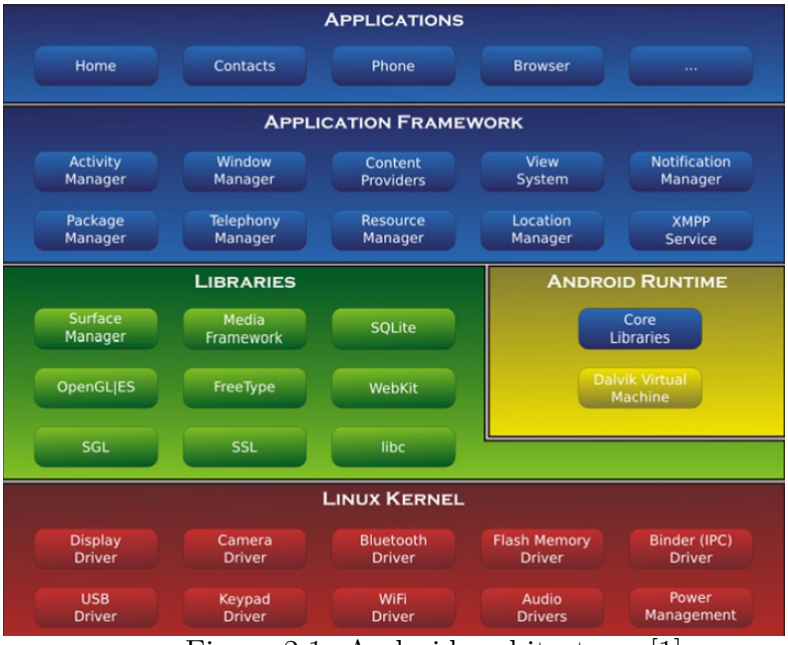


Figure 2.1: Android architecture. [1]

At Android runtime, there is the Dalvik VM, that is a optimised version of java virtual machine (JVM). For security reasons, each app runs in separate processes in its own Dalvik VM instance [30]. Running separately, apps can communicate with the Android framework or with other apps just using Android inter-process communication (IPC) mechanism [30]. Therefore, apps are isolated from each other and can not access data from another app. However, there are other ways for apps to communicate with each other. For instance, the communication via messaging objects called Intents, that allow communication between services, activities and broadcast, via messages sent between components. To understand an app behavior and issues related to app security, it is crucial to analyse this Intent-based communication.

The application framework layer includes Activity Manager, that manages the life-cycle and interaction of the activities running on the system. It also includes Window Manager that forwards UI (User Interface) input to the app, and allows applications to draw on the screen.

On top of the Android architecture, there are Android Applications (or *apps*), such as games, browsers and contacts. Apps are written using programming languages that compile for the Java Virtual Machine (JVM) bytecode, such as Java or Kotlin. The JVM bytecode is compiled to a Dalvik Executable (DEX) bytecode format, and is then packaged along with other application resources to form an Android Application Pack (APK) file, which can be distributed for end-users. The app building process can be seen at Figure 2.2.

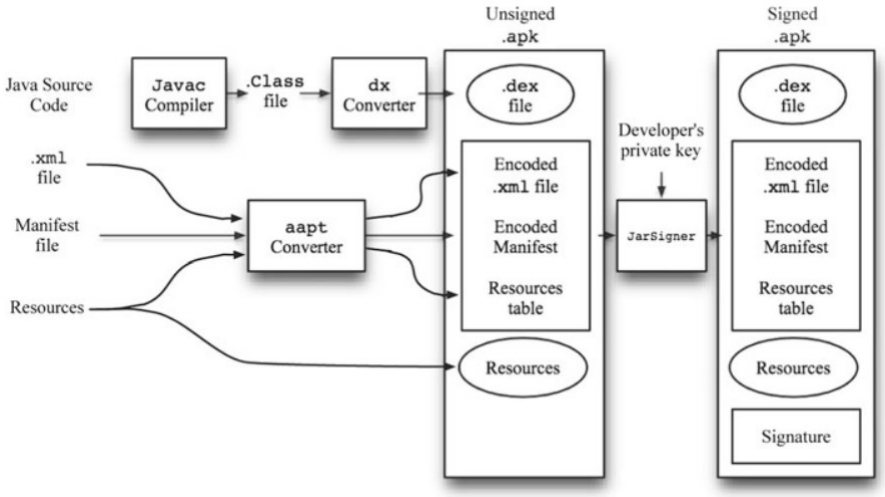


Figure 2.2: Android build process, from source code to APK file. [2]

2.1.2 Android Permission System

In order to access information beyond the application sandbox [1], apps must acquire permissions and then access the resources through specific system Application Programming Interfaces (APIs). Android has two main types of permissions [31]: install-time permissions, and runtime permissions. Install-time permissions are less sensitive permissions and must be requested upon installation of the APK on the system, such as internet access. Runtime permissions are highly sensitive permissions and are generally related to users' private data, such as camera or microphone access. Runtime permissions must be acquired right before the actual use of the resource, and are only available on Android 6.0 (API 23) or higher [32].

Google Play Store is the primary market source for Android apps and has a flexible policy regarding the process of publishing apps. Therefore, every month many Android apps are cleared from it because of issues related to spyware and other types of malware [33]. For security reasons, Google Play lists each app with the requested permission, and those permissions have been presented to the user during the installation process. This process can be canceled, if the user does not feel comfortable with the requested permission.

[1] More details about sandbox at Section 2.3

This security procedure can ensure control over user sensitive data access, and reduce vulnerabilities problem in Android apps. However, it can be inefficient if app developer requests more permissions than they really need. This overprivileged problem often occurs because of developer error, which many times feels confused about the permission system. This uncertainty can lead to overprivileged apps, in a development effort to make the apps work accurately [29].

Common overprivilege problem can occurs when the developer request permissions that sound correlated to their apps purpose [29]. For example, assuming that an app need access status of all networks. There are two permissions that have similar sounding names: **ACCESS_NETWORK_STATE** and **ACCESS_WIFI_STATE**. The developer can uncertainly request them in pairs, even when just one is demanded, in our example: **ACCESS_NETWORK_STATE**, leading to unnecessary access to WIFI permissions.

Although developers may ask for unnecessary permissions due to confusion or forgetfulness, it is possible that this problem can be intentional, since malicious developers could also create overprivileged apps (malware) in order to stealthily access users' sensitive resources [34]. According to Zhou et al [35], malicious apps tend to requests surely more permissions than benign ones. In their studies, they notice that on average, malicious app request 11 permissions, while normally benign apps request just 4.

Beyond the unnecessary permissions issues, Android permission system are not security indicators that guarantee fulfill its purpose, since it does not help users make good security decisions. Some studies point out that Android users are careless, and do not pay enough attention to or understand permission warning [36], considering that they just want the end product. However, the same studies also show that there are Android users that demonstrate awareness and understanding of permissions, proving that permissions can help some users to avoid privacy-invasive apps. Furthermore, these experts users could protect other Android users since they can write negative reviews when they find unneeded permission requests. Among study participants, 24% pointed out that they had relied on reviews to get better permission information.

2.2 Android Malware

Android developers have developed apps every day that cover a growing range of functionalities [5]. Unfortunately, the increasing of Android apps also comes with rapid growth of security threats and a variety of attacks. [7], bringing a critical need to effectively mitigate them. Because of the variety of attacks, a good understanding of mobile malware is needed to develop an effective solution.

Zhou et al. [35], presented a systematic characterization of existing Android malware. They categorized existing strategies Android malware used to install onto users' phones, and generalized them into three main techniques no mutually exclusive: *repackaging*, *update attack* and *drive-by download*.

According to Zhou et al., *repackaging* is the most common strategy malware developers use to piggyback malicious code into original apps. They found 86% of malware are repacked apps, demonstrating the popularity and gravity of *repackaging* strategy. Basically, malware authors download an original and popular app from the official Android market, disassemble the app, enclose malicious code at original, re-assemble and submit the "malicious" apps to alternative Android markets, or/and to official market [37]. Since the application code is compiled to a DEX bytecode, some characteristics about the original code such as class names, method names, and types are retained, which makes them more susceptible for reverse engineering if compared to native code [38]. Android app users could be vulnerable when installing these modified apps. More details in the next subsection.

Update attack is a strategy very similar to *repackaging*, but more difficult for detection [39]. As the first strategy, it also repackages a benign app, however this time, instead of enclosing the malicious code as a whole, it only includes an update component. When the malicious app runs, the user will be prompt that a new version is available. If the user update the app, an "updated" version with malicious code will be installed, enclosing the "update" version inside the original. As the malware extracts malicious payload from the external environment, it is more furtive, and static code analysis techniques may fail to detect them [35].

This last strategy is a traditional *drive-by download* attack. Android users are motivated to download interesting and apparently useful apps, but in fact they are "wolf in sheep's clothing". Apps that promise to improve battery efficiency, or claims to better protect online banking operations are examples of this strategy. However, these "useful" apps can be very harmful to your users, causing serious problems, like leak sensitive banking information or sending of SMS message to a premium-rate number.

2.2.1 Repackaged Android Apps

There are many tools available that help developers reverse engineer Android bytecode [40]. For this reason, software developers can easily decompile trustworthy apps, modify their contents by inserting malicious code, repackage them with malicious payloads, and re-publish them in app stores, including official ones like the Google Play Store. It is well-known that repackaged Android apps can leverage the popularity of real apps to increase their propagation and spread malware [10]. As an example, in 2016 a repackaged

version of the famous Pokémon Go app was discovered less than 72 hours after the game was officially released in the United States, Australia, and New Zealand [10]. The repackaged version, originated from an unofficial app store, gained full control over the victim's phone, obtaining access to main functions such as the phonebook, audio recorder, and camera.

Repackaging has been raised as a noteworthy security concern in Android ecosystem by stakeholders in the app development industry and researchers [41]. Indeed, there are reports claiming that about 25% of Google Play Store app content correspond to repackaged apps [42]. Nevertheless, all the workload to detect and remove malware from markets by the stores (official and non-official ones), have not been accurate enough to address the problem. As a result, repackaged Android apps threaten security and privacy of unsuspicious Android app users, beyond compromising the copyright of the original developers [43]. Aiming at mitigating the threat of malicious code injection in repackaged apps, several techniques based on both static and dynamic analysis of Android apps have been proposed, including the MAS approach for malware classification [13, 3].

2.2.2 Android Malware Detection Systems

Malware has always been a constant concern surrounding Android apps. New malware is being developed every day, so staying up to date is pretty hard, since most antivirus solutions work with known virus signatures. Many Android devices have been dealing with malware over the years, since many apps are deployed to Play Store from a wide range of different sources, making it harder for security researchers to cover most of the apps. Since the Android policy for apps is flexible, there is need for a solution that quickly analyzes and isolates new software that might be flagged as malicious.

As far as malware detection goes, many actions were taken to minimize the risk of malware infection in Android apps. Many open source platforms share known malware samples with researchers [44]. Those samples help antivirus softwares get rid of most of common malware, since they need to know the malware behavior to successfully block them.

The field of malware detection for the Android platform is fertile, with a significant number of secondary studies already published [45, 46, 47, 48]. In general, malware detection techniques are divided into static detection, dynamic detection, and hybrid detection [49, 50]. Several studies have also conducted surveys on malware detection techniques and presented a review of them [51, 52, 53]. For instance, M. Odusami et al. [53] discuss various static analyses approaches that have been used in the literature to identify malicious behavior in Android apps. The authors present some works with permission and signature-based malware detection systems. They highlight that both approaches have

a low false positive rate; however, they are very ineffective in detecting new malware. Although they could reveal possible malicious behaviors, the authors discuss several limitations of these approaches, as they are limited regarding code obfuscation and dynamic code loading.

Some existing malware detection techniques in Android systems are surveyed and compared using metrics such as base of detection, analysis technique, accuracy, evaluation metric, operation level, classifier/tool, output of system and scope of improvement[50]. These are some example of malware detection systems that are described at [50]: DREBIN [54], AndroDialysis [55], ICCDetector [56], APK Auditor [57], DroidNative [58], DMDAM [59] and API based system [60].

The literature also presents surveys based on dynamic analysis, exposing risks that are not detected by static analysis. As a malicious app "is alive", dynamic analysis adds another degree of analysis since it observes how Android apps interacts with the environment. However, if applied inappropriately, it may provide limited code coverage, which repeated executions can improve. Therefore, dynamic analysis's time cost and computation resources are higher when compared with static analysis. K. Tam et al. [52] presented several dynamic analysis studies based on Android architectural layers. The survey also exposed that dynamic analysis can be performed in emulator environments, real devices, or both. The authors discuss that the choice of environments is an important issue for analysis, as there are malware families that can detect emulated environments and do not exhibit malicious behaviors [61]. Finally, K. Tam et al. also exposed some works based on hybrid malware detectors and claim that these methods can increase code coverage and robustness, taking advantage of the best of each technique to find malicious behaviors.

Several studies have also explored Android malware detection approaches based on machine learning (ML) techniques, employing both static and dynamic analyses to extract features and train ML models [51]. Most of these approaches have demonstrated high accuracy (above 90%), effectively detecting previously unseen malware families with low false positives rate [62]. However, some studies have identified limitations of machine learning approaches for Android malware classification. In their work, K. Liu et al. [51] highlighted challenges related to machine learning techniques, identifying several factors that could lead to biased results, such as the quality of the sample set. The authors argue that samples of poor quality, with a non-representative size or outdated samples, may yield promising results in experimental settings but might not perform similarly in a real environment [51]. Another critical aspect is the quality of the extracted feature dataset. The efficacy of machine learning approaches heavily relies on the selection of correct features and their extraction methods, particularly dynamic features. Moreover,

in addition to the computational costs involved, other studies [63, 64] have indicated that machine learning approaches exhibit weaknesses when dealing with malicious apps that alter their behavior to mislead learning algorithms, which may restricts their applicability in real-world scenarios.

2.2.3 Android Security Concerns

Overall security has always been a concern on Android devices and it is widely known by researchers that Android has always been an easy way to spread malware around the world. Since mobile is one of the most used vehicles of communication and concerning security, it is poorly used by its end-users. To put it into perspective, at December 2011 Google Play exceeded 400000 apps, doubling in quantity in just 8 months [39], and it is known by any security researcher that the weakest point of interaction on any software is the user. The challenge on most papers resides in enhancing Android security and ensuring that its users have your sensitive resources protected [65].

Taking it to the next level of security is an approach that many scientists and researchers are doing by experimenting with sandboxes to test Android apps [13]. This improves overall security, although it is known that there's much more progress to be made to catch up with malware creators.

2.3 Sandbox

A sandbox is an isolated environment on an electronic device within which apps cannot affect other programs outside its boundaries, like the file system, the network, or other device data [66]. It enables testing and execution of unsafe or untested code, possible malware, without worrying about the integrity of the electronic device that runs the app [67]. This need may arise in a variety of situations, such as when running software input by untrusted users, in malware analysis, or even as a security mechanism in case a trusted system gets compromised [66]. A sandbox environment should be able to shield the host machine or operating system from any damages caused by third party software. Thus, sandbox environment should have the minimum requirements to run programs (make sure the program will not impact resources outside the sandbox), and make sure it will never assign the program greater privileges than it should have, working with the principle of the least privilege, giving permissions to users according to their needs, *i.e.*, giving them no more power than needed to successfully perform their task. This principle prevents escalating privileges and unauthorized access to resources, thereby improving the system's overall health.

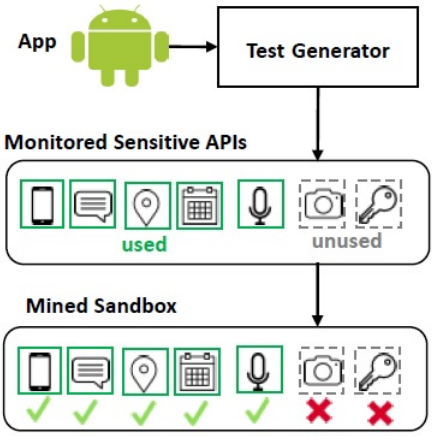


Figure 2.3: Mining Sandbox [3]

2.3.1 Sandbox and Android Security

As Android OS is based on Linux, every app is initialized as a separate process and executed in the same manner. With this architecture, the principle of least privilege is realized through sandboxing process, where apps never access the data of other apps, and an app just accesses user resources, like contacts and location, through specific APIs (Application Programming Interface), which are in-turn guarded by permissions.

Nowadays, malware becomes more stealthy and hackers learn how to avoid anti-virus signature checks, by obfuscating the native code [68] , calling java libraries, or even by creating root exploits, since the native code is allowed to directly make syscalls. Another approach to rooting is making a side attack aimed at a benign app to make a syscall, exploiting its native code to get root access. Those are kinds of attacks that are usually tackled with by Android Mining Sandbox [13].

2.3.2 Mining Android Sandbox

The Android Mining Sandbox concept is a sandboxing technique that consists of mining rules from an Android app, and use these rules to ensure system security. The technique was first proposed by Jamrozik et al. [13] and comprises of two steps. First, rules are mined and will compose the sandbox, through test generator tools, that identify sensitive Android API, called during their execution. These tools investigate program behavior through resource accesses realized by these APIs. The second stage ensures that resources not accessed, or accessed differently from the first-stage, are not accessible by the apps. So, if a malicious app requires access to resources, different from what was previously

mined, the sandbox will prohibit this access. The Figure 2.3 (taken from the paper [3]), shows the core ideal of this technique.

Automatically mining software resources or components, to infer behavior is not new and has been discussed before. For instance, Whaley et al. [69] combine dynamic and static analysis for API mining and so extract interface from software components. Ammons et al [70] propose a machine learning approach, called specification mining, to discover temporal and data-dependence relationships that a program follows when interacting with an API or abstract data type.

The main purpose of test generation tool is to find bugs. However, it also can be used to explore program behavior, and thus assist in the task of building Sandboxes. Regarding test generating tools used for mining Sandboxes, Jamrozik et al [16] proposed DroidMate, a test generation tool that implements a pseudo-random GUI exploration strategy, and was the first approach to leverage test generation to extract sandbox rules from apps. Li e tal. [20] proposed DroidBot, a test generator tool that explores sensitive resources access by apps, following a model-based exploration strategy. In their work, the authors present a comparison between DroidBot and Monkey [17] regarding malware analysis, and showed that Droidbot can trigger a number of sensitive behaviors, like data leaks and file accesses, higher than Monkey. From the same authors, another test generator tool for Android, described as Humanoid [71], is a Droidbot evolution and presents a proposal that can generate humans like tests inputs, using deep learning.

2.3.3 Mining Android Sandbox for Malware Classification

Besides being used to generate Android sandboxes, the MAS approach can also be used to detect if a repackaged version of an Android app contains an unexpected (perhaps malicious) behavior [3]. In this scenario, the *effectiveness* of the approach is estimated in terms of the accuracy in which malicious behavior is correctly identified in the repackaged version of the apps.

The MAS approach for malware classification typically works as follows. In a first step (**instrumentation phase**), a tool instruments the code of the apps (both original and repackaged versions) to collect relevant information during the apps execution in later stages. Then, in a second step (**exploration phase**), the MAS approach collects a set S_1 with all calls to sensitive APIs the original version of an app executes while running a test case generator tool (like DroidBot). In the third step (**exploitation phase**), the MAS approach (a) collects a set S_2 with all calls to sensitive APIs the repackaged version of an app executes while running a test case generator tool, and then (b) computes the set $S = S_2 \setminus S_1$ and checks whether S is empty or not. The MAS approach classifies the repackaged version as a malware whenever $|S| > 0$.

Previous research works reported the results of empirical studies that aim to investigate the effectiveness of the MAS approach for malware classification [3, 24]. For instance, Bao et al. found that, in general, the sandboxes constructed using test generation tools classify at least 66% of repackaged apps as malware in a dataset comprising 102 pairs of apps (original/repackaged versions) [3]. Actually, the mentioned work performed two studies: one pilot study involving a dataset of 10 pairs of apps (`SmallE`), in which the authors executed each test case generation tools for one hour; and a larger experiment (`LargeE`) involving 102 pairs of apps in which the authors executed the test case generation tools for one minute [3].

The authors also presented that, among five test generation tools used, DroidBot [20] leads to the most effective sandbox. Le et al. extend the MAS approach for malware classification with additional verification, such as the values of the actual parameters used in the calls to sensitive APIs [21], while Costa et al.[26] investigated the impact of static analysis to complement the accuracy of the MAS approach for malware classification. Their study reports that DroidFax [23], the static analysis infrastructure used in [3], classifies as malware almost half of the repackaged apps.

2.4 Static and Dynamic Code Analysis and Detect Attacks

Static code analyzing is straightforward, it pretty much extracts most of the software patterns to analyze it without executing it. Due to the app is not required to be executed, the static approach is resource and time-efficient. Provide information about permissions, API calls, .dex files for opcodes, and metadata are some examples of what is gathered with this type of analysis for further investigation [5]. Decompilers are often used too since they try to rebuild the code by the execution flow based on assembly instructions. There are frameworks to help researchers dealing with such activities.

As far as Dynamic code analysis is concerned, it monitors the execution flow of any app in search of any malicious behaviour while the app is running. Dynamic analysis can monitor, for instance, network traffic, CPU utilization, battery usage, and API assess [5], in a controlled environment. Some known techniques are applied to enhance detection, like try to detect misuse of PII (Personal Identifiable Information) of users. Another approach that researchers use is monitoring system calls, which some of them can be very critical in some cases. Along with syscalls monitoring, Java method invocations, for instance, are usually monitored, since it's easier to invoke dangerous functions that can do anything with the infected device.

Many of these techniques were initially developed for malware-targeting desktop systems, but they have since been adapted and extended to address threats in other environments, such as mobile platforms. For the sake of brevity, on this section we focus on the most closely related work, particularly advancements in network flow analysis and machine learning (ML) algorithms for malware classification. At the end, we also present an overview of some of these approaches, described at secondary studies [72, 51] (Table 2.1).

2.4.1 Dynamic and Static Analysis Approaches

Dynamic approaches, like TaintDroid [73], DroidRanger [74], and DroidScope [75], monitor app behavior in real-time, offering high accuracy but significant performance overhead, limiting their practical use on mobile devices. In contrast, static methods such as Kirin [76], Stowaway [29], and RiskRanker [77] are efficient and scalable but heavily rely on manually defined patterns, hindering their effectiveness against novel malware. In addition, these methods often lack transparency, making it difficult to understand their decision-making processes. The lack of understanding about the combination of both techniques is also evident in Costa et al. [78], which presents an empirical study showing that this combination enhances detection capabilities by leveraging both types of information to identify and analyze malware more effectively.

2.4.2 Network Traffic Analysis

Recently, increasing attention has been directed toward malicious network traffic generated by suspicious apps. As a result, researchers have begun to analyze and identify malicious applications based on their network traffic. For example, Wang et al. [79] have proposed an Android malware detection method based on HTTP flow analysis, using Natural Language Processing (NLP) techniques. Researchers have also explored the automatic generation of network signatures [80, 81, 82], often focusing on worm identification. For instance, Perdisci et al. [83] propose an approach that generates network signatures for mobile malware by analyzing similarities in HTTP traffic and clustering malicious patterns. Although effective against known threats, signature-based methods struggle to detect novel attacks due to their reliance on predefined patterns.

Some studies utilize text analysis based on Packet/Flow textual features for malware detection. For instance, Nan et al. [84] introduced UIPicker, a framework that uses NLP, ML, and program analysis to identify personal user information on a large scale. Recon et al. [85] recently proposed a method to detect and prevent personal information leaks in mobile network traffic by analyzing key-value pairs. Some authors have also used

Packet/Flow features statistically. Arora et al. [86] compared malware traffic with benign network traffic, identifying deviations in network behavior using statistical features like average packet size, flow duration, and byte ratios. AppScanner [87] is a framework that uses encrypted network traffic statistical features to automatically fingerprint and identify malicious apps. Conti et al. [88] analyzed encrypted Android traffic to identify user actions based on statistical features.

2.4.3 Machine Learning Approaches

ML approaches have been used in network traffic-based malware detection methods [89, 90]. Depending on the type of ML model used, these methods can be divided into two categories: (1) shallow learning techniques and (2) deep learning techniques [91].

Shallow techniques are traditionally used in malware detection as a classification problem. Previous research [90] developed a detection system that monitors network data and other key features, such as battery consumption and temperature, to detect malware with an accuracy of over 85.6%, using statistical and ML methods. Martín et al. [92] introduced CANDYMAN, a malware classification tool that uses Markov Chains to model normal network traffic patterns and identify anomalies that may be associated with malicious activities. The study employed deep learning techniques and the Random Forest algorithm, reporting a detection accuracy of 77% and 81.8%, respectively. The authors used a dataset of 4,442 samples from 24 different malware families.

Lashkari et al. [93] analyzed network traffic features to distinguish malicious traffic from normal traffic and classify apps as malware or non-malware, using common classifiers such as logistic regression, decision trees, random forests, and KN. Feizollah et al. [94] also analyzed network traffic using a similar set of ML classifiers, including KNN, SVM, decision trees, Naïve Bayes (NB), and multi-layer perceptron (MLP). The authors confirmed that KNN delivered the best performance among these classifiers. Garg et al. [95] proposed a network-based detection model for Android malicious apps. This method collects network activity features from 18 different malware families and 14 legitimate apps and uses them to detect Android malware using traditional ML techniques. The authors reported an accuracy between 95% and 99%. Dash et al. [96] introduced DroidScribe, a framework that employs a dynamic analysis approach to classify malware by monitoring network transactions generated by apps. Similarly, another study [97] introduces DynaLog, a dynamic analysis method for malware classification that relies on API calls and service executions. We summarize this related work on Table 2.1.

Although we attempted to replicate some of the results of these related studies, it was not feasible because they do not provide replication packages. Moreover, most of the studies are based on limited datasets, in terms of both size and the lack of recurrent

malware families—preventing a proper evaluation of their results. For instance, references [92, 95, 96, 89] have a large number of samples, but they cover only 24, 18, 23 and 5 malware families, respectively (see Table 2.1). Considering these two limitations, that is, the lack of a replication package and the use of questionable datasets, we decided to design a new approach (Chapter 6) and conduct a comprehensive study that differs from the related work in several key aspects. First, due to the challenges associated with manually generating input test cases, we employ an automated test case generation tool to simulate user interactions with apps within a short time frame [25]. While some studies have also utilized test case generation tools for input generation [92, 96], they primarily rely on *Monkeyrunner* tool—a native tool provided by the Android SDK for stress testing, which operates on a random strategy. In our study, we use the *Droidbot* [20] tool, a more advanced, open-source test generation tool for Android apps. Several studies [3, 20] highlight that DroidBot outperforms other test generation tools that rely on random strategies, demonstrating a greater capability in uncovering a larger number of potential malicious behaviors. Second, we considered a more representative set of 2,886 malware from 116 families. Also, unlike previous research, we allow full reproduction of our work, publishing Python scripts and datasets used as replication packages[2].

Table 2.1: Studies based on network data analysis.

Reference	Technique	Algorithms	Samples	Replication package	F1-score
[90]	ML	RF, SVM, LMT	200	NO	85.6%
[92]	ML, DL, Markov chains	SVM, RF, CNN, LSTM, RNN	4,442	NO	81.8%
[93]	ML	RF, KNN, DT, RT, RL	400	NO	91.41%
[94]	ML	SVM, MLP, DT, KNN, NB	100	NO	99%
[86]	ML	DT	27	NO	93.75%
[95]	ML	DT, LR, KNN, BN	1,260	NO	99%
[96]	ML	SVM	4,533	NO	94%
[89]	ML	BN, LR, RF, SVM	3,526	NO	99%

2.5 Taint Analysis

Android apps contain within themselves the risk that sensitive data, such as credit card details, device ID, contacts can be leaked into public sinks like the internet [98].

Taint analysis is a special type of static or dynamic analysis that purpose is to track sensitive data within programs [99]. It identifies sensitive information leakage detecting *taint flow* between *source* and *sink*. In Android apps context, a data leak occurs when sensitive data, such as contact, or device ID, flows from a private source to public sinks, like the internet. Taint analysis can present possible malicious data flow to malware detection

[2]https://github.com/droidxp/ML

tools or even for a human check, that can decide if the “source-sink" relationship is or is not an unwanted behavior. Thereby, taint analysis monitor sensitive source “tainted" through the app by starting at a pre-defined point.

In Android context, sources are the APIs in which apps access sensitive information, called *sensitive* APIs. The analysis follows the data flow until it reaches a sink, like a method that sends SMS. Finally, it brings exact information about which data will be leaked and where [11]. The Android SDK provides APIs that allow apps to send private data to other apps on the same device, or remote devices. As these APIs may lead to sensitive data leakage, they are security-critical and require special attention and control [73]. (Listing 2.1) presents a simple data leakage example. In this example, the device information is captured at line 4 (source) and then leaked at line 9 (sink), by SMS transmission.

Listing 2.1: Simple Data Leakage

```
1 > localObject2 = (TelephonyManager)getSystemService("phone");
2 > if (localObject2 != null)
3 > {
4 >  this.imei = ((TelephonyManager)localObject2).getDeviceId();//source
5 > }
6 > if ("".equals(this.destMobile)) {
7 >  getDestMobile();
8 > }
9 > sendSMS(this.destMobile, "imei:" + this.imei)//sink
```

Wei et al. [100] propose a scalable taint analysis for Android apps that applies traditional taint analysis techniques with targeted optimizations specific to Android OS. Flowdroid [11] improves on precision of traditional approaches by including context and flow sensitivity. A significant issue with taint analysis is the cost of the tool itself hampering the performance. FastDroid [101] mitigates this issue by introducing an intermediate light-weight abstraction to perform the analysis.

2.6 Symbolic Execution

Symbolic execution is an elegant software analysis solution introduced over 40 years. It arises from the need of check if certain properties can be violated by a piece of software [102][103], like array index out of range exception, division by zero, or any security violation.

The main idea is to explore many possible execution paths at the same time without requiring concrete inputs. Instead of taking on specific input values, symbolic execution abstractly represents them as symbols and simultaneously explores multiple paths, that a program could take under different inputs. It has proved to be useful in many settings like mission-critical and software security, leading to breakthroughs at software reliability [4].

To exemplify the use of symbolic execution, consider a simple method show in (Listing 2.2). It was written at java language and return a double value, through 2 integer inputs. Our idea is to determine which inputs make the method *twoIntToDouble* fail at line 8, caused by divide-by-zero error.

Listing 2.2: Method that return a float through 2 integers. Adapted from [4]

```
1 > double twoIntToDouble (int a, int b) {
2 >     int x = 2, y = 0;
3 >     if (a != 0) {
4 >         y = 4 + x;
5 >         if (b == 0)
6 >             x = 2 * (a + b)
7 >     }
8 >     return x/(x-y);
9 > }
```

Considering that each input parameter can take 2^{32} distinct integer values (4 bytes), a concrete execution would be very expensive, since that runs concretely the method *twoIntToDouble* on randomly generated input, will spend much time. Symbolic execution can solve this problem by evaluating the code using symbols, and reason on classes of inputs, instead of concrete input values.

Figure 2.4 shows a “execution tree" that represent the execution paths followed during the symbolic execution of the method *twoIntToDouble*. At the beginning, input arguments (a and b) are associated with symbolic values (α_a and α_b), and path constraints are true (π = true). Then, at node 2, the local variables x and y are initialized, updated with concrete value ($x \rightarrow 2$, $y \rightarrow 0$), and symbolic stored at “α". At method *twoIntToDouble*, line 3, there is the first conditional statement execution, and the node 2 has two arcs leaving, which are labeled “$\alpha_a \neq 0$" and “$\alpha_a = 0$" for the true (then) and false (else), respectively. Depending on the branch taken, new assumptions are associated to symbol π ($\pi = \alpha_a \neq 0$ or $\pi = \alpha_a = 0$) at node 3 and 4 respectively. At node 5, there is a second conditional statement execution, and the two arcs leaving, are labeled with “$\alpha_b = 0$" and “$\alpha_b \neq 0$" for the true (then) and false (else). Variable y is also updated with $4 + x$, becoming $y \rightarrow 6$, since $x \rightarrow 2$ at node 3. At node 8, variable x is updated with $x = 2 * (a + b)$ from note 6, becoming $x \rightarrow 2 * (\alpha_a + \alpha_b)$.

Once all branches reaches the line 8 of *twoIntToDouble* method (return x/(x-y)), the technique follow analysing which input values for parameter a and b can violate software integrity because of divide-by-zero error. Analyzing nodes 4, 7 and 8, we can deduce that only node 8 can reach values in which $(x - y) = 0$. In general, if there is any input value, that make the final sentence true, these input will make *twoIntToDouble* method break. Thus, at node 8 the final follow sentence has a input value that makes it true. For instance a = 3 and b = 0.

$$2 * (\alpha_a + \alpha_b) - 6 = 0 \wedge \alpha_a \neq 0 \wedge \alpha_a = 0 \Longleftrightarrow (\alpha_a = 3 \wedge \alpha_b = 0)$$

Thus, symbolic execution has proved to be an accurate technique for code analysis. However, it is also known to suffer from severe challenges when dealing with real codes, with more complex objects than presented in our example, like arrays and pointers, for instance. It also can be problematic when requires modeling their interactions with sur-

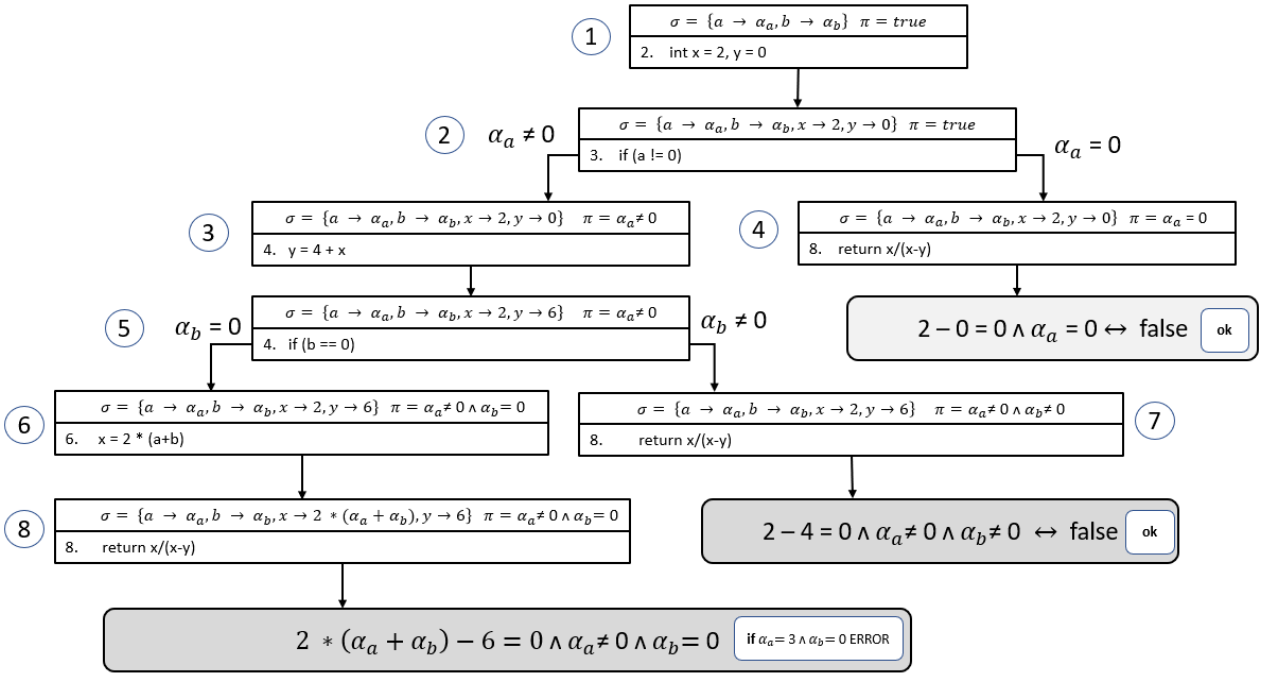


Figure 2.4: Symbolic Execution tree. Adapted from [4]

Chapter 3

DroidXP

A Benchmark for Supporting the Research on Mining Android Sandboxes

This chapter corresponds to our paper published in the 20th International Working Conference on Source Code Analysis and Manipulation (SCAM), Adelaide, SA, Australia, 2020, pp. 143-148.
doi: 10.1109/SCAM51674.2020.00021.
(F. H. da Costa et al.)

3.1 Introduction

The Android operating system is the most widely used mobile platform, dominating the marketplace of smartphones, tablets, and others electronic devices. Due to this popularity, the number of incidents related to Android malicious software (malware) has significantly increased [7, 8]. In this context, security issues in Android apps have become a relevant research topic, and many techniques have been developed to identify malicious software and vulnerabilities in Android apps [8]. For instance, researchers have explored the use of dynamic analysis tools and test case generators to mine sandboxes, whose goal is to protect Android users from malicious behaviors [13].

The main idea of mining sandboxes is to explore the set of calls to sensitive API methods, using test case generation tools. These tools, explore apps behaviors based on their sensitive APIs. A sandbox builds upon the calls to these sensitive Android APIs, during the execution of test cases (exploratory phase). The sandbox could then block future calls to other sensitive resources, which diverge from those found in the exploratory phase. Using this approach, the more efficient the test generating tool is (for instance, in

terms of code coverage), the more accurate should be the sandbox (in terms of malware identification).

Previous research studies have investigated the effectiveness of mining sandboxes, using automated testing tools, to compare the performance of test generation tools to mine sandboxes. For instance, Bao et al. [3] present the results of an empirical study comparing 5 automated testing tools (DroidMate, Monkey, GUIRipper, Puma, and Droidbot) for mining sandboxes. The results show that DroidBot was more efficient to detect malware in a dataset comprising 102 pairs of benign and malign apps. Nonetheless, the field of automatic test generation of mobile apps is everything but listless—many promising automatic test generation tools have been recently proposed [106, 107], which might outperform previous tools that have been used to mine sandboxes. For instance, Humanoid generates *human like* test inputs using deep learning techniques, and empirical studies suggest that it leads to a best code coverage than all the other aforementioned tools [71].

That is, along with the research advancements in the area of test case generation, new solutions are constantly emerging, and researchers and practitioners (including ourselves) have come up against technical difficulties in to reproduce the previous studies that compare test case generation tools to mine sandboxes. We argue that this problem is mostly due to the lack of a benchmark support, that can assist researchers and practitioners in this timely task.

As an approach to mitigate this problem, here we introduce DroidXP, a benchmark to help researchers and practitioners to integrate test case generation tools and compare their performance on mining sandboxes to the Android platform.

3.2 DroidXP Principles: A Benchmarking Metaphor

Benchmarking is the activity of measuring and evaluating the relative performance of an asset (e.g., an algorithm or tool) against the performance of another asset—considering well-defined conditions [108]. Therefore, the objective is to make comparisons between different solutions that share a similar purpose, or between different versions of the same System Under Test (SUT). To make the comparisons, some measurement instruments are necessary; to collect a set of metrics that are relevant for a given domain. DroidXP has been designed to compare the performance of test generation tools to mine Android sandboxes.

According to Bouckaert et al. [108], one can collect primary and secondary metrics while benchmarking computational assets. Primary metrics are those collected directly from the SUT, and secondary metrics are those concerning the environment in which

the SUT is operating. In our benchmark, we are mostly interested in two primary: test coverage and number of detected malwares.

In every benchmarking process, a well-defined setup is a fundamental concern that must be considered before assessing the SUTs. For instance, in the context of DroidXP, our setup corresponds to (a) the corpus of Android apps (pairs of benign/malicious apps we use in the analysis), (b) the maximum execution time of each test case generation tool (the systems under test), and (c) the number of repetitions in the experiments. These elements are sufficient to evaluate several scenarios, which can support the research on mining sandbox. The response variables are the means we use to compare the SUTs. As mentioned, we consider two response variables: test coverage and number of malwares each tool detects.

Benchmarks also take into account another important aspect, which brings more accuracy for the research activity: comparability. Since test generation tools rely on non-determinism, the different executions of a tool (repetitions) in the same settings might produce different outcomes (in terms of code coverage and malware detection). Since we have to produce a *close result* for comparisons, we use the *average* of the response variable collected on a number of n repetitions.

3.3 DroidXP Benchmark

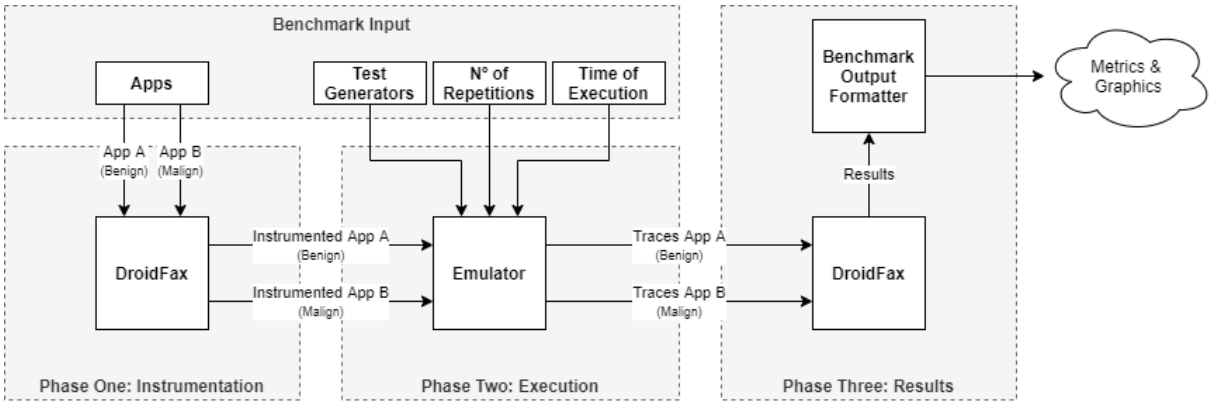


Figure 3.1: Benchmark architecture

To mitigate those problems, we first conducted a domain engineering to identify the main requirements for using the test case generation tool. The results of this domain engineering, together with the principles that we discussed in the previous section, guided the DroidXP implementation decisions.

For instance, DroidXP relies on a simple *Command Line Interface* (CLI) that favors the execution and configuration of the benchmark. DroidXP also relies on DroidFax [23],

a tools that instruments Android apps and collects relevant information about their execution (using the test case generation tools). DroidFax already collects code coverage information and the set of sensitive APIs a given app calls during a test execution.

We implemented DroidXP using the Python programming language (version 3). The reasons for using Python include: (a) a rich set of libraries for writing CLIs, for simplifying the process of system calls, and for implementing data analysis; (b) support for object-oriented constructs and reflection (mechanisms we used to create the DroidXP extension points); and (c) the familiarity the researchers had with the language

The DroidXP CLI provides two commands: a command that lists all test case generation tools (executing the project with the option "list-tools") that had been integrated into the benchmark; and a command that performs the execution of the benchmark, which can be configured using the following parameters:

- `-tools`: Specifies the test tools used in the experiment
- `-t`: Specifies the threshold (in seconds) for the execution time in the experiment
- `-r`: Specifies the number of repetitions used in the experiment
- `-output-format`: Specifies the output format
- `-debug`: Specifies to run in DEBUG mode (default: false)

The DroidXP architecture relies on the pipes-and-filters architectural style [109] (Figure 3.1), and includes three main components; where each component is responsible for a specific phase of the benchmark (instrumentation, execution, and result analysis).

3.3.1 Phase 1: Instrumentation

In the first phase, a researcher must define the corpus of APK files DroidXP should consider during a benchmark execution. After that, DroidXP starts the DroidFax service that instruments each APK file, so that DroidXP would be able to collect data about each execution. To improve the performance of the benchmark, the instrumentation phase runs only once for each APK. In this phase, the DroidFax tool also runs some static analysis procedures, to collect the number of methods and classes of each app, which is a necessary information to estimate code coverage.

3.3.2 Phase 2: Execution

In this phase, DroidXP installs an (already instrumented) APK file into an Android emulator, and then executes a test case generation tool during a period of time. This

process repeats for every test case generation tool and APK files. To provide repeatability of the experiment, DroidXP removes all data stored in the emulator before starting a new execution. That is, every execution uses a *fresh* emulator, without any information that might have been kept during previous executions.

We designed the benchmark so that it is relatively easy to add new test case generation tools. To achieve this goal, we leverage the Strategy Design pattern [110], which sets a contract between a family of classes that, in our case, abstracts the specificities for running each tool we want to integrate into DroidXP. Listing 3.1 shows the contract a developer must override to integrate a new tool. According to this contract, one have to: (a) implement a class that inherits from `ToolSpec`, define constructors that calls the `super` constructor, passing the name, the description, and the process id as arguments, and (c) implement the `execute` method, which receives as argument the path of an APK file and a timeout.

Listing 3.1: ToolSpec Sample

```
class ToolSpec(AbstractTool):
  def __init__(self):
    super(ToolSpec, self).__init__(
      "Test Generator Name",
      "Test Generator Description",
      "process_id"
    )

  def execute(self, path, timeout):
    # Execute test generator ...
```

DroidXP uses the last parameter to kill the execution process on the emulator after a *timeout event* throws. This step is necessary to provide a fresh environment for the next test execution. The actual logic for executing a specific tool resides in the `execute` method, so it is the responsibility of a developer to set up all configuration necessary to run a specific test case generation tool. We have already integrated the following tools into DroidXP.

(a) **Monkey** is a Google testing utility that generates pseudo-random streams of user events.

(b) **DroidBot** is a test input generator for Android, which sends random or scripted input events [20].

(c) **DroidMate** is an automated execution generator / dynamic analysis engine for Android apps [16].

(d) **Sapienz** is a multi-objective approach to automated input test generation for Android [111].

(e) **Humanoid** is a tool that uses deep learning techniques to explore Android apps, mimicking the human behavior [71];

3.3.3 Phase 3: Result Analysis

During the execution, all the data that is required to compute the results are provided by Logcat [112], one of the Android SDK's native tools and is a command-line tool that dumps a log from the Android emulator. Thus, the only part of the log that is analyzed in this phase is the messages sent by the methods inside the Android app that were instrumented on the first phase using the DroidFax tool.

Droidfax computes the coverage each test achieved and which sensitive API was accessed during the execution of that test. That last information is required to compute the test generator performance in identifying malicious apps by spotting differences between the sensitive API accessed by each version of an app. This information is vital to the measurement of the test generator performance and qualification. After this phase, the benchmark outputs the results of the experiment, which is informing the performance of one or more testing generator tools in mining sandboxes.

3.4 Empirical Study

3.4.1 Study Settings

This empirical study aims to reproduce a previous research work on mining sandboxes [3] and to experiment with the DroidXP usage. Similarly to the previous study, here we also leverage two metrics to compare the performance of test case generation tools: code coverage and the number of detected malware. For code coverage, we actually use the percentage of application methods each tool traversed—during the execution phase. We also use the same pairs of apps of the previous study—a corpus that contains a sample of 102 pairs (benign/malign) of apps from Androzoo [113].

We investigated the following questions in our study:

(RQ1) What is the performance of each tool in terms of the number of detected malware?

(RQ2) What is the strength of the correlation between code coverage and the number of detected malware?

(RQ3) What is the relationship between the coverage rate and the accuracy of each tool in detecting malicious apps?

Similarly to the work of Bao et al. [3], we conducted two experiments. In the first, we executed the benchmark considering all 102 apps in our corpus, and an execution timeout of one minute. We used the first experiment to answer the first research question (RQ1). In the second, we considered a subset of the apps in our corpus, comprising only 10 apps; and executed the benchmark in three different configurations of timeout: 1 minute, 5 minutes, and 10 minutes. We used this second experiment to answer the questions RQ2 and RQ3.

3.4.2 Study Results

Figure 3.2 summarizes our findings with respect to the first research question. All tools were able to correctly identify at least 40% of the malwares. Interesting, in our study, two recent tools were introduced, and its efficiency was presented compared to other existing tools: Sapienz (42%) and Humanoid (56%). These tools have not been considered in the previous work. Nonetheless, we realized a different performance of the tools DroidMate, DroidBot, and Monkey; in comparison with the results of the previous study [3]. As a future work, we will investigate the possible reasons for this difference.

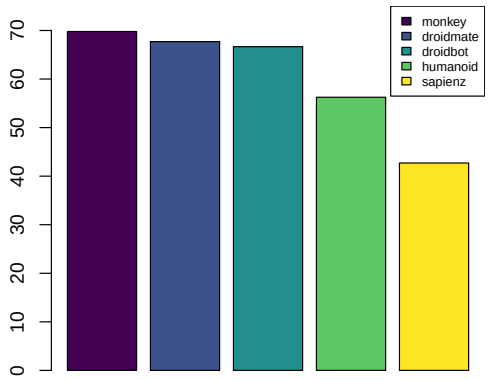


Figure 3.2: Summary of the percentage of malware correctly detected.

Figure 3.3 summarizes our findings addressing the second research question (RQ2). We realized two interesting findings: first, Monkey and Humanoid outperform the remaining tools in terms of code coverage, when executing in the first two configuration: *Config-(a)* (1 minute) and *Config-(b)* (5 minutes). Surprisingly, Humanoid led to a smaller code coverage for *Config-(c)* (10 minutes) than for *Config-(b)*. Moreover, we did not realize

any benefit of increasing the timeout from five minutes to 10 minutes, when considering the code coverage criteria.

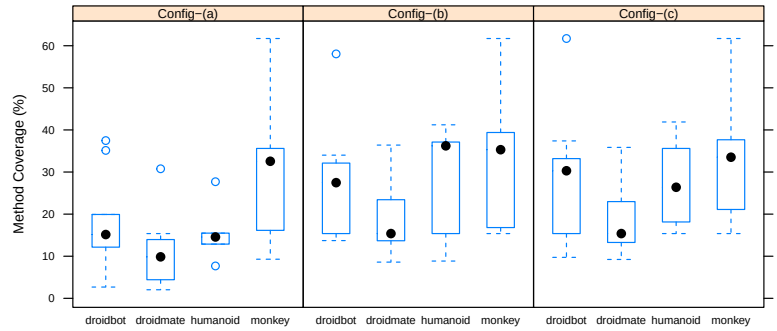


Figure 3.3: Summary of method coverage over the tools.

Finally, we answered our last research question (RQ3) using the Spearman correlation test. In this way, we estimated the strength of the correlation between method coverage and number of detected malwares. The results reveal a moderate (p-value = 0.68), negative correlation ($\rho = -0.31$) between these measurements. That is, improving the performance of code coverage might not directly contribute to the performance of detecting malware.

3.4.3 Discussion and Limitations

Our empirical study allowed us to experiment with DroidXP. We successfully integrated five different tools and reproduced a previous research work [3]. Nonetheless, some of our results differ from the work by Bao et al. [3]. This might have occurred because either we could not reproduce the same settings of the previous work or DroidXP somehow impacted the performance of the tools. We will investigate this issue as a future work. Another limitation of our study is that DroidXP did not collect the code coverage metric for the Sapienz tool. It is not clear to us the reason for this problem. However, we were not able to collect Sapienz coverage even when executing it apart from DroidXP.

3.5 Related Work

To the best of our knowledge, this is the first work that proposed a Benchmark tool for test case generation. However, many works present a comparative study of the main existing test input generation tools for Android [25][3]. Furthermore, other papers [20][107][114][16][111] providing a set of automated input generators that could be measured and analyzed by our proposed tool. Li et al. proposed in [20], DroidBot,

a UI-guided test input generator for Android apps, that support model-based test input generation. In this paper, the authors show the comparison between DroidBot and Monkey[17] in a proof of concept example of using DroidBot in malware analysis, which could have been easily accomplished in our benchmark tool. From the same authors, other GUI test input generator for Android, described as Humanoid [71], is a Droidbot evolution and presents a proposal that is able to generate humans like tests inputs, using deep learning. This work presents comparisons between state of the art testing tools, and presents higher tests coverage results. [114] Propose Stoat, another model-based test input generation. The paper presents comparisons between 3 other tools, exposing coverage line, bugs and crashes statistics, but does not focus on malware detection capabilities. [111] Proposed Sapienz, a test input generator tool that performs static and dynamic analysis and conducts a benchmark study, between 2 other tools, presenting crashes detected and coverage rate. All these studies are possible to be reproduced at DroidXP, which has an easier integration with other tools that emerge.

3.6 Final Remarks

The DroidXP tool is a first attempt to create a benchmark tool to assist researchers and practitioners in developing new test generating tool solutions, for mining Android sandboxes, by providing metrics and graphics that bring performance comparison against other. A case study over five test generator tools, including two new tools, proved that DroidXP is a good solution, to compare test generation tools. In our solution was proposed tree extensions, where it is possible integrate new tools for analysis simply, is possible change the Apps set to be analyzed, and it is possible choose report type for wished format. As future work, we plan extend the project for new issue analysis and extend the benchmark, not just focused on test generation tools for mining Android sandboxes, but test generation tools in general. We also plan extend the benchmark to test the efficiency of the combination of two or more test case generation tools. The source code of DroidXP can found at GitHub [115].

Chapter 4

Dissecting the MAS approach

Exploring the Use of Static and Dynamic Analysis to Improve the Performance of the Mining Sandbox Approach for Android Malware Identification

This chapter corresponds to our paper published in the Journal of Systems and Software, Volume 183, 2022, 111092, ISSN 0164-1212.
doi: https://doi.org/10.1016/j.jss.2021.111092.
(F. H. da Costa et al.)

4.1 Introduction

Almost two-thirds of the world use mobile technologies [116], and the Android Operating System has dominated the market of smartphones, tablets, and others electronic devices [117]. Due to this growing popularity, the number of incidents related to Android malicious software (malware) has significantly increased. In only three years, researchers reported a substantial increase in the population of Android malware: from just three families and a hundred samples in 2010 to more than a hundred families with thousands of samples in 2013 [7, 8]. Security issues in Android software applications [1] have become a relevant research topic, and many techniques have been developed to identify vulnerabilities in Android apps [11], including the use of static analysis algorithms either to identify privacy leaks or to reveal the misuse of cryptographic primitives [14, 15], for instance.

Another alternative for protecting users from Android malicious behavior consists in the use of dynamic analysis to mine Android sandboxes [13]. The mine sandbox approach

[1] In this chapter, we will use the terms Android Applications, Android Apps and Apps interchangeably, to represent Android software applications

starts with an exploratory phase, in which a practitioner takes advantage of automatic test case generator tools that explores an Android application while recording the set of sensitive APIs the app calls. . This set of senstivie calls comprises a sandbox infrastructure. After the exploratory phase, the sandbox might then monitor any call to sensitive APIs while a user is running the app, blocking the calls that have not been identified during the exploratory phase—thereby protecting Android users from additional malicious behavior [13]. Jamrozik et al. argue in favor of dynamic analysis for mining sandboxes, instead of using static analysis—mostly because of the overapproximation problem: "static analysis often assume that more behaviors are possible than actually would be" [13]. In addition, code that uses dynamic features (such as reflection) poses additional challenges to static analysis algorithms—even though *dynamic features* of programming languages are often used to introduce malicious behavior. Even though these claims are reasonable, previous research results do not present empirical assessments about the limitations of static analysis to mine sandboxes. Consequently, it is not clear whether and how both approaches (dynamic and static analysis) could complement each other in the process of mining Android sandboxes.

The lack of understanding about static and dynamic analysis complementing each other also appears in the work of Bao et al. [3] (hereafter BLL-Study), which presents an empirical study that explores the performance of dynamic analysis for identifying malicious behavior using the mining sandbox approach.

Their study leverages DroidFax [23] to instrument 102 pairs of Android apps (each pair comprising a benign and a malicious version of an App) and to collect the information needed to mine sandboxes (that is, the calls to sensitive APIs). Although the authors report a precision of at most 70% of dynamic analysis tools to differentiate the benign and malicious versions of the apps, the authors ignore the fact that DroidFax statically analyzes the Android apps and also records calls to sensitive APIs (besides instrumenting the apps). As we discuss in this chapter, this DroidFax static analysis component leads to an overestimation of the performance of the dynamic analysis tools for mining sandboxes and might have introduced a possible threat to the conclusions of that work. In the security domain, overestimating the performance of a technique for malware identification brings serious risks, and we show here that DroidFax inflated significantly the performance of the dynamic analysis tools for mining sandboxes, as reported in the BLL-Study.

The goal of this chapter is two fold. First we present the results of an external, non-exact replication [118] of the BLL-Study. To this end, we take advantage of DroidXP, a tool suite that helps researchers (including ourselves) to integrate test case generation tools and compare their performance on mining Android sandboxes. We discussed the design and implementation of DroidXP in a conference paper [24], which also includes

an initial evaluation of DroidXP. As a matter of fact, the results of the first DroidXP evaluation revealed a possible overestimation in the performance of dynamic analysis tools as reported in the BLL-Study—which in the end motivated us to conduct the non-exact replication of that study. Here we extend our previous work with a couple of customizations of DroidXP, which allowed us to reproduce the BLL-Study by means of a serie of new experiments that reveal the actual performance of the dynamic analysis tools. Section 4.2.1 revisit the DroidXP design, while Section 4.2.2 discuss the setup of our replication study.

Second, in this chapter we also explore how a static analysis approach (based on taint analysis) compares and complements the mining sandbox technique for identifying malicious behavior that infects benign applications. The idea here is to compare the dataflows from *source* to *sink* statements computed using two executions of the FlowDroid infrastructure [22]: one execution that analyses a benign version of an Android app and one execution that analyses a malicious version. We consider that the taint analysis approach is able to identify a malware whenever we find a dataflow from a source to a sink in the second execution that does not appear in the first one. We detail the settings of this taint analysis study in Section 4.2.3

Altogether, this chapter brings the following contributions:

- A replication of the BLL-Study that better clarifies the performance of dynamic analysis tools for mining Android sandboxes. The results of our replication (Section 4.3.1) give evidence that the previous work overestimated the performance of the dynamic analysis tools—that is, without DroidFax (an independent component used for running the BLL-Study experiment), the performance of the tools drop between 16.44% to 58%.
- A broad comprehension about the role of static analysis tools for mining sandboxes, showing that we can benefit from using both static and dynamic analysis for detecting malicious Android apps. In addition, we give evidence that a well known static analysis approach, based on taint analysis, leads to a performance similar to the dynamic analysis approach for diferenciating benign and malicious versions of the same app (Section 4.3.2).
- A reproduction package of our study that is available online, including scripts for statistic analysis [2] and tooling for reproducing and extending our study. The repository for DroidXP is available at GitHub[3].

[2]https://github.com/droidxp/paper-replication-package
[3]https://github.com/droidxp/benchmark

4.2 Study Settings

Our research work aims to better understand the use of static analysis to mine Android sandboxes and explore the benefits of combining taint analysis with the mine sandbox approach for identifying malicious behavior. On the one hand, Jamrozik et al. suggest that a static analysis approach for mining sandboxes might be ineffective—due to *over-approximation problem* [13]. However, to the best of our knowledge, there is no empirical study comparing static and dynamic analysis for mining sandboxes. On the other hand, the BLL-Study explored the mining sandbox approach by comparing the performance of five **dynamic analysis tools** (DroidMate, DroidBot, PUMA, GUIRipper, and Monkey) for identifying malicious behavior. Nonetheless, their research also involved an external static analysis component (DroidFax) whose impact on the results was not measured—in terms of malware identification. This lack of understanding about the implications of static analysis for mining sandboxes motivates our research, which investigates the following research questions.

(RQ1) What is the impact of the DroidFax static analysis algorithms on the results of the BLL-Study? We estimate the impact in terms of the number of detected malwares.

(RQ2) What is the effective performance of each sandbox, in terms of the number of detected malware, when we discard the contributions of the DroidFax static analysis algorithms?

(RQ3) What are the benefits of using taint analysis algorithms to complement the dynamic analysis approach for mining sandboxes, in terms of additional malwares identified?

Answering the research questions RQ1 and RQ2 allows us to better understand the relevance of combining static and dynamic analysis for mining Android sandboxes. Moreover, exploring RQ1 and RQ2 can reveal a possible overestimation of the performance of the dynamic analysis tools in the BLL-Study. Answering research question RQ3 allows us to open up the possibility of finding new strategies for malware detection, complementing the performance of dynamic analysis through the use of static analysis algorithms. We conducted two empirical studies to answer the research questions above. We address the research questions RQ1 and RQ2 in the first empirical study, whose goal is to conduct a non-exact replication of the BLL-Study. We conduct the first empirical study using DroidXP [24] (Section 4.2.1), a tool that simplifies the execution of experiments that compare the performance of dynamic analysis tools in the task of identifying malwares, using a mining sandbox approach. We present the study settings of the first empirical

study in Section 4.2.2. In the second empirical study we use FlowDroid [11] to investigate the suitability of taint analysis algorithms to complement the mining sandbox approach for identifying malwares, and thus it targets our third research question (RQ3). We present the settings of the second empirical study in Section 4.2.3.

4.2.1 The DroidXP benchmark

We designed and implemented DroidXP to systematically assess and compare the performance of test generation tools for mining android sandboxes. It allows the integration and comparison of test case generation tools for mining sandboxes, and simplifies the reproduction of the studies. DroidXP relies on a simple *Command Line Interface* (CLI) that simplifies the integration of different test generation tools and favors the setup and execution of the experiments. DroidXP also relies on DroidFax, which instruments Android apps and collects relevant information about their execution, including the set of sensitive APIs a given app calls during a test execution. DroidFax also collects inter-component communication (ICC) using static program analysis.

The DroidXP CLI provides commands for listing all test case generation tools (executing the project with the option "list-tools") that had been integrated into the tool and commands for executing the experiments. An *experiment run* can be configured according to several parameters, including:

- `-tools`: Specifies the test tools used in the experiment
- `-t`: Specifies the threshold (in seconds) for the execution time in the experiment
- `-r`: Specifies the number of repetitions used in the experiment
- `-output-format`: Specifies the output format
- `-debug`: Specifies to run in DEBUG mode (default: false)
- `-disable-static-analysis`: Disable DroidFax static analysis phase (default: false)

Figure 4.1 shows the DroidXP architecture, based on the pipes-and-filters architectural style [109]. The architecture includes three main components; where each component is responsible for a specific phase of the experiments execution (instrumentation, execution, and result analysis).

Phase 1: Instrumentation

In the first phase, a researcher must define the corpus of APK files DroidXP should consider during an experiment execution. After that, DroidXP starts the DroidFax service

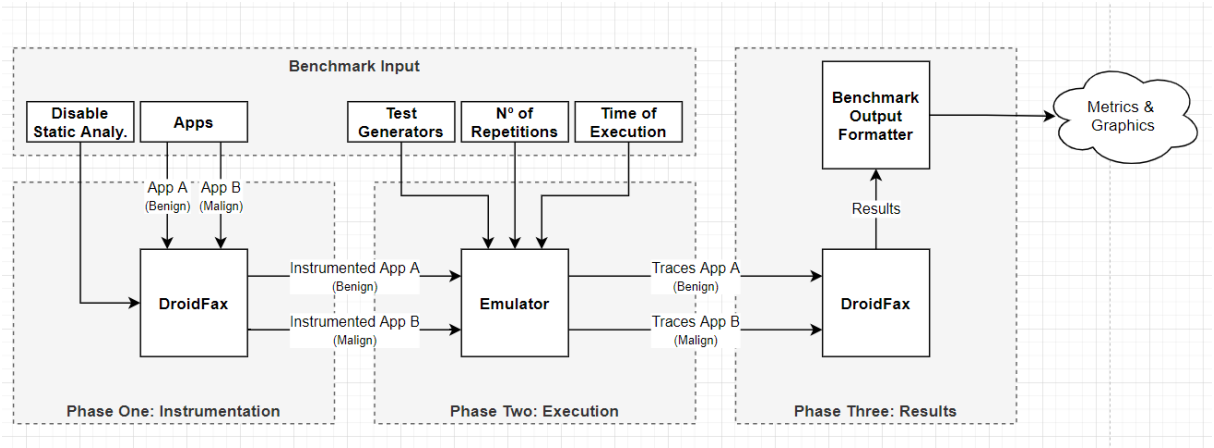


Figure 4.1: DroidXP architecture

that instruments each APK file, so that DroidXP would be able to collect data (e.g., calls to sensitive APIs) about each execution. To improve the performance of DroidXP, the instrumentation phase runs only once for each APK. In this phase, the DroidFax tool also runs some static analysis procedures—when the option `-disable-static-analysis` is not set.

Phase 2: Execution

In this phase, DroidXP installs an (already instrumented) APK file into an Android emulator, and then executes a test case generation tool during a period of time. This process repeats for every test case generation tool and APK files. To provide repeatability of the experiment, DroidXP removes all data stored in the emulator before starting a new execution. That is, every execution uses a *fresh* emulator, without any information that might have been kept during previous executions. It is relatively easy to add new test case generation tools into DroidXP. Indeed, every new tool must override two methods of a `Tool` abstract class (according to the Strategy Design pattern [110].

Phase 3: Result Analysis

During the execution of the instrumented apps, all data that is relevant to our research is collected by Logcat [112], one of the Android SDK's native tools. Logcat dumps a log from the Android emulator while the already instrumented app is in execution. The part of the log we analyze in this phase comprises the data sent by the methods within the Android app that were instrumented on the first phase using the DroidFax tool.

This data includes method coverage from the execution of each test generator tool and the set of sensitive APIs the app calls during its execution. This set of calls to sensitive

APIs is necessary to estimate the test generator performance in identifying malicious apps—by spotting differences between the sensitive API accessed by each version of an app (benign or malign). In the end, DroidXP outputs the results of the experiment, which gives the performance of one or more testing generator tools in mining sandboxes.

We used the DroidXP infrastructure to conduct our first empirical study, whose settings we present in the following section.

4.2.2 First Study: A replication of the BLL-Study

The BLL-Study reports the results of an empirical study that compares the performance of test generation tools to mine Android sandboxes [3]. Since the BLL-Study does not compute the possible impact of DroidFax into the performance of the test generation tools, here we replicate their work to understand the impact of the DroidFax static analysis algorithms into the BLL-Study results.

Our replication differs from the original work in a few decisions. First, here we isolate the effect of the DroidFax static analysis algorithms, in the task to identify malicious apps. In addition, although we use the same dataset of 102 pairs of Android apps used in the BLL-Study, here we discarded 6 pairs for which we were not able to instrument—out of the 102 pairs used in the original work, originally shared in the AndroZoo repository [113]. We also introduced a recent test generator tool (Humanoid [71]), which has not been considered in the previous work. Finally, we extended the execution time of each test generation tool, executing each app from the test generation tool for three minutes (instead of one minute in the original work), and built the sandboxes after executing each test generation tool three times—the original work executed each test generation tool only once. It is important to note that our goal here is not to conduct an exact replication of the BLL-Study, but instead understand the role of the DroidFax static analysis algorithms in the performance of test case generation tools for mining sandboxes.

Besides Humanoid, our study considers three test generation tools used in the BLL-Study: DroidBot [20], DroidMate [16], and Monkey [17]. We selected DroidBot and DroiMate because they achieved the best performance on detecting malicious behavior—when considering the 102 pairs of Android apps (B/M) in the BLL-Study. It is important to note that here we used a new version of DroidMate (DroidMate-2), since it presents several enhancements in comparison to the previous version. We also considered the Google's Monkey open source tool, mostly because it is the most widely used test generation tool for Android [119]. Monkey is part of the Android SDK and does not require any additional installation effort. We included Humanoid in our study because it is a recent tool that emulates realistic users, creating human-like test inputs using deep learning techniques.

Data Collection

Similarly to the BLL-Study, besides method coverage information, our experiments record every call to *sensitive methods* of the Android platforms, while a given test case generation tool is running. We consider the same set of 97 sensitive methods from the AppGuard privacy-control framework uses [120].

We executed DroidXP using two configurations. In the first (named WOS), we executed DroidXP using the dataset of 96 pairs of Android apps—each pair including a benign and a malign version, the four test case generation tools (DroidBot, DroidMate, Monkey, and Humanoid), and the `-disable-static-analysis` option of DroidFax, which disables the execution of the DroidFax static analysis component from the experiment. The WOS configuration runs the test case generation tools for three times, using a time limit of three minutes. In the second configuration (named WS), we executed DroidXP using the same dataset of 96 pairs of Android apps, though also executing a fake test case generator tool (named Joker) **without** the `-disable-static-analysis` option. Joker simulates a test tool that does not run the Android apps during an experiment execution, and its usage allow us to estimate the actual performance of the DroidFax static analysis component.

Using the WS configuration, the Execution Phase of DroidXP does not collect any call to sensitive APIs, and thus we can estimate the performance of the static analysis component of DroidFax (answering RQ1). Differently, the WOS configuration disables the static analysis component of DroidFax and we could better estimate the true performance of the test case generation tools for mining android sandboxes (answering RQ2). For comparison purpose, we also executed the four test case generation tools using the DroidFax static analysis component.

Data Analysis

DroidXP produces a dataset with the sensitive APIs that the benign / malign versions of an app call, during the execution of each test case generation tool. We estimate the performance of a test case generation tool by considering the percentage of malwares in our dataset its resulting sandbox is able to identify .

Recall that we build a sandbox during the exploratory phase of the mining sandbox approach. This exploratory phase records the set of sensitive APIs a benign version of an app calls—during the execution of a test case generation tool. Similarly to the BLL-Study, we consider that a sandbox of an app identifies a malware whenever the malicious version makes a call to a sensitive API that has not been recorded during the exploratory phase (see Figure 4.2).

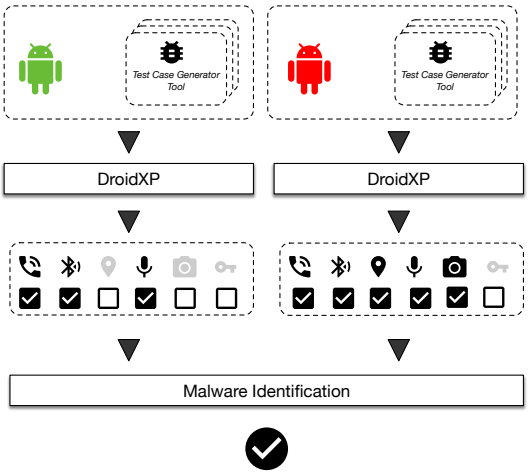


Figure 4.2: Overview of our approach for malware identification in the first study.

To sum up, in order to analyse the performance of the test case generation tools (including Joker), we just have to compare the calls to sensitive APIs made by the benign and malign versions of the apps, during the execution of the tools. In the end, we generate a set of observations, where each observation contains the tool name, the number of the repetition (in the range [1..3]), a boolean value reporting the use of the DroidFax static analysis component, and a boolean value indicating whether or not the malware has been identified. We use descriptive statistics and plots to compare the performance of the tools and answer RQ1 and RQ2. We also use *Logistic Regression* [121, Chapter 4] to understand the statistical relevance and the contribution of each feature (tool, repetition, DroidFax static analysis component) to malware identification. Our hypothesis here is that the DroidFax static analysis component has a positive effect on the performance of the sandboxes to identify malwares.

4.2.3 Second Study: Use of Taint Analysis for Malware Identification

In the second empirical study we investigate whether or not a taint-based static analysis approach is also promising for identifying malwares, given a version of an app that we can assume to be secure (goal of research question RQ3). To this end, we leverage the FlowDroid taint analysis algorithms for Android apps (version 2.8), in order to identify dataflows that might lead to the leakage of sensitive information. Our goal here is to investigate if it is possible to detect malicious behavior by means of the *divergent* source-sink paths that FlowDroid reveals after analysing a benign and a malign versions of an Android app.

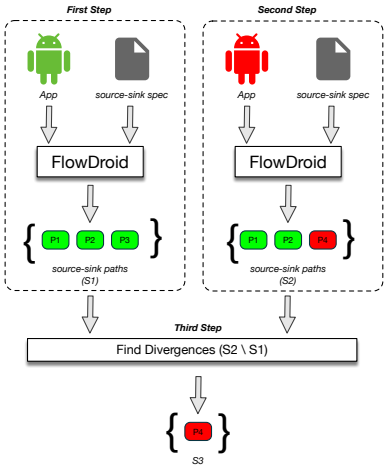


Figure 4.3: Overview of our approach in the second study.

Data Collection

FlowDroid takes as input an Android Application Package (APK file) and a set of API methods marked either as **source** or **sink** (or both). Source methods are those that access *sensitive information* (e.g., a method that access the user location), while sink methods are those that *might share information with external peers* (e.g., a method that sends messages to a recipient). We rely on the source-sink definitions of the FlowDroid implementation [22, 122], which involves a curate list of source and sink methods (including callbacks and other Android API methods of interest). FlowDroid then uses a *context, flow, and field sensitive analysis* to identify dataflow paths from sources to sinks [22].

Our data collection approach involves three steps (see Figure 4.3). In the first, we execute FlowDroid to mine the source-sink paths from a benign version of an app, and then enumerate a set (S1) with the possible dataflows between sources and sinks. All paths in S1 are considered secure in our analysis. In the second step we repeat the FlowDroid execution, though considering the malicious APK version of the app. This leads to a second set (S2) of source-sink paths.

It is important to note that not all source-sink paths are malign, and then we follow a specific methodology to identify malwares using taint analysis. That is, we only report a malware when FlowDroid finds an additional source-sink path in the malicious version of an app, which has not been identified when analysing the benign version. Therefore, in the third step we compute the difference (S3) between the sets S2 and S1 (i.e., $S3 = S2 \setminus S1$). If the set S3 is not empty, we assume that FlowDroid has identified the malware.

In this second study we use the same dataset of 96 pairs of Android apps (B/M) used in the first empirical study.

Data analysis

We use two metrics in this second study: the total number of malicious apps FlowDroid is able to find and the execution time for running the taint analysis algorithm for each app. Similarly to the first empirical study, we use descriptive statistics and plots to compare the performance of the taint analysis and mining sandbox approaches. We also use *Logistic Regression* [121, Chapter 4] to better understand the statistical significance of the benefits of using FlowDroid (in comparison to the DroidFax static analysis component only). Our hypothesis here is that FlowDroid outperforms, in terms of the number of detected malware, the sandbox generated by the DroidFax static analysis component.

4.3 Results and discussion

In this section we detail the findings of our study. We present the results of the first and second studies in Section 4.3.1 and Section 4.3.2, respectively. In Section 4.4 we summarize the implications of our study.

4.3.1 Result of the first study: A BLL-Study replication

Our first study is a replication of the BLL-Study. As discussed in the previous section, we first executed the analysis using the DroidXP benchmark with its default configuration. Then we repeated the process, however this time we isolate the effect of the static analysis component of DroidFax. In this way, we could better estimate the performance of the dynamic analysis tools for mining Android sandboxes. Table 4.1 summarizes the results of the executions. The columns Exec. (WS) and Exec. (WOS) show the number of malwares identified when executing each tool with the support of the DroidFax static analysis algorithms (WS) and without the support of DroidFax static analysis algorithms (WOS). The Impact column shows (in percentage) to what extent the DroidFax static analysis algorithms influences the performance of the sandboxes created after executing the test generation tools. We calculate the impact using Eq. (1).

$$Impact \quad = \quad \frac{(Exec.\ (WS) - \ Exec.\ (WOS)) \times 100}{Exec.\ (WS)} \tag{4.1}$$

Table 4.1 shows that the impact of DroidFax in the results is significant, ranging from 16.44% (DroidBot) to 51.79% (Humanoid). Note that, in the BLL-Study, the authors do not present a discussion about the influence of DroidFax in the performance of the test generation tools, even though this influence is not negligible. Considering the Joker

tool, our fake test generation tool that does not execute the apps during the benchmark execution, DroidFax improves the performance in 100%. This result is expected, since the Joker tool does not execute any dynamic analysis. Next we discuss the result of each individual test generation tool.

Table 4.1: Summary of the results of the first study.

Tool	Exec. (WS)	Exec. (WOS)	Impact (%)
DroidBot	73	61	16.44
Monkey	71	56	21.13
DroidMate	68	52	23.53
Humanoid	56	27	51.79
Joker	42	0	100.00

DroidBot in the first execution (Exec. WS) led to a sandbox that detected a total of 73 malware among 96 pairs present in our dataset (76.04%), detecting more apps with malicious behavior than any other tool. Similar to the BLL-Study, DroidBot is the test case generation tool whose resulting sandbox detected the largest number of malicious apps. Moreover, in our second execution (Exec. (WOS)), removing the DroidFax static analysis support reduced the DroidBot performance in 16.44%, the smaller impact we observed among the tools.

Monkey in the first execution (Exec. (WS)) produced a sandbox that detected 71 out of the 96 pairs of Android apps. Contrasting, in the original study, the Monkey's sandbox detected 48 malwares within the 102 pairs (47.05%). This difference might be due to the fact that Monkey uses a random strategy for test case generation and here we considered the outcomes of three executions—while in the BLL-Study, the authors consider the outcomes of one execution. Considering our second execution (Exec. (WOS)), there is a reduction of 21.13% in the Monkey's performance, leading to a sandbox that was able to detect 56 malwares.

DroidMate in the first execution (Exec. (WS)) led to a sandbox that detected 68 apps with malicious behavior (70.83%). In the BLL-Study study, DroidMate also detected 68 malwares, though considering the 102 pairs of apps used in the original study. In the second execution (Exec. (WOS)), without the DroidFax static analysis algorithms, the resulting sandbox's performance drops by 23.53%, being able to detect 52 out of the 96 pairs of Android apps.

Humanoid showed the worst performance, even though a previous work [71] presented that it leads to the highest number of lines coverage in comparison to Monkey,

DroidBot, and DroidMate. This might suggest that, since Humanoid learn how humans interact with apps, and use the learned model to guide test generation, at simulate environment, this method to generate test inputs are less effective to build Android sandbox, in comparison with techniques that rely on random testing (such as Monkey). In the first execution (Exec. (WS)), the resulting Humanoid sandbox identified 56 malwares in our dataset (58.33%). Humanoid was the most affected in the second execution (Exec. (WOS)), whose resulting sandbox presents a reduction of 51.79% in the number of detected malwares. Since the BLL-Study did not explore Humanoid, we do not have a baseline for comparison with the previous work.

Joker is our fake test case generation tool that help us understand the performance of the DroidFax static analysis algorithm for mining sandboxes. We integrated Joker into the DroidXP benchmark as an additional test case generation tool that does not run the Android apps. As a result, the analysis using Joker reveals the performance of DroidFax static analysis algorithms alone. For the first execution, with the DroidFax static algorithms enabled, even though Joker does not execute the Android apps, its resulting sandbox detected 43.75% of the malwares. For the second execution, that is, disabling the DroidFax static analysis algorithm, the resulting Joker sandbox was not able to detect any malware. Therefore, our results show that DroidFax alone is able to detect more than 40% of the malicious version of the apps.

Finding 1 *Integrating the dynamic analysis tools with the DroidFax static analysis algorithms improves substantially the performance of the resulting Android sandboxes for detecting malicious behavior.*

The Venn-diagram of Figure 4.4 summarizes how the tools can complement each other. Note in the diagram that 53 malwares have been detected by all sandboxes generated in the first execution (with the DroidFax static analysis algorithms), out of the 78 malwares identified by at least one sandbox. In addition, the DroidMate sandbox did not detect any malware that had not been detected by the other tools. Differently, the Monkey sandbox detected three malwares that had not been detected by any other sandbox, the DroidBot sandbox detected two malwares that had not been detected by any other sandbox, and the Humanoid sandbox detected one malware that had not been detected by any other sandbox. Contrasting with the BLL-Study, our results suggest that using DroidMate in combination with Monkey, DroidBot, and Humanoid does not improve the general performance of an integrated environment for mining Android sandboxes.

Finding 2 *Our results suggest that one might benefit from using an integrated environment that combines Monkey, DroidMate, and Humanoid to mine Android sandboxes. Contrasting with the BLL-Study, introducing the DroidMate tool does not improve the overall performance for detecting malwares using a mining sandbox approach.*

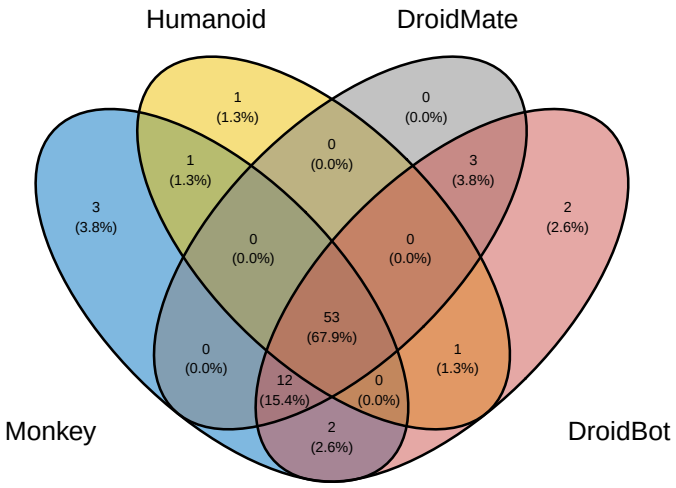


Figure 4.4: Venn Diagram highlighting how the sandboxes from the tools can complement each other.

Altogether, ignoring Joker, our study reveals that from 58.33% (Humanoid) to 76.04% (DroidBot) of the malicious apps investigated in our study can be detected using the sandboxes generated after running the test case tools with the support of the DroidFax static analysis algorithms. We also investigate if the use of the DroidFax static analysis component leads to a statistically significant benefit on malware identification. To this end, we build a logistic regression model in the form $Malware \sim Tool + StaticAnalysis + Repetition$. Table 4.2 shows the results of the logistic regression analysis, highlighting that (a) Humanoid has a negative, though significant impact on malware identification; and (b) the use of DroidFax static analysis has a positive and significant impact on malware identification.

Besides that, in the first execution (WS), none of the resulting sandboxes could detect 18 malwares in our dataset (18.75%). According to the Euphony tool [123], 12 of these 18 malwares are *adwares*, 3 are *trojans*, 2 are PUPs (*Potentially Unwanted Program*), and one is an *exploit*. At this point, an additional question arises: what are the characteristics of the malwares that have (not) been identified using the mining sandbox approach? To explore this question, we take advantage of the `dex2jar` tool to reverse-engineer all 96 malwares considered in our analysis and computed the *diffs* of the benign/malicious

Table 4.2: Results of the Logistic Regression (first study)

	Estimate	*p-value*	*C.I.*
Tool [DroidBot]	0.1034	0.4718	(-0.133,0.340)
Tool [DroidMate]	-0.0561	0.6955	(-0.292,0.180)
Tool [Humanoid]	-0.8910	0.0000 ***	(-1.131,-0.651)
Tool [Monkey]	-0.0110	0.9390	(-0.247,0.225)
DroidFax static analysis	0.8867	0.0000 ***	(0.743,1.031)
Repetition	-0.0171	0.7487	(-0.105,0.071)
AIC	3001.07		
Num. obs.	2304		

*** *p-value* < 0.001

versions of the APPs. The results of this activity are available in our replication package.[4] In what follows we dissect a few examples of malwares that at least one of the resulting was able to identify. After that, we present the characteristics of a malware that none of the sandboxes was able to detect. Our goal here is to provide a lower-level intuition about the classes of malware the mining sandbox approach is not able to detect. A reader that is not interested in these details could skip to Section 4.3.2.

To start with, consider the malicious version of the app `com.andoop.flyracing`—which both DroidBot and Humanoid sandboxes could detect in our analysis. In this particular case, the malicious version changes the Android Manifest file, adding permissions to receive and send SMS messages (Listing 4.1). Adding these permissions, a malicious app may get money fraudulently by sending messages without user confirmation, for instance. The pair `L:M` indicates a code segment that appears in line `L` of the malicious (`M`) version of an app.

After decompiling this malware, we also observed that the malicious version of the `MainService` class introduces a behavior that collects sensitive information (the International Mobile Equipment Identity, IMEI) and sends it using an SMS message (Listing 4.2).

Listing 4.1: Diffs in the `com.gau.screenguru.finger` AndroidManifest file of the malicious version

```
67:M >    <uses-permission android:name="android.permission.RECEIVE_SMS"/>
68:M >    <uses-permission android:name="android.permission.SEND_SMS"/>
```

Listing 4.2: Diffs in the malicious version of the class `com.android.main.MainService` (app `com.gau.screenguru.finger`)

```
492:M > localObject2 = (TelephonyManager)getSystemService("phone");
493:M > if (localObject2 != null)
494:M > {
```

[4] https://github.com/droidxp/paper-replication-package/blob/master/diff/

```
495:M >  this.imei = ((TelephonyManager)localObject2).getDeviceId();
496:M >  this.imsi = ((TelephonyManager)localObject2).getSubscriberId();
497:M >  this.iccid = ((TelephonyManager)localObject2).getSimSerialNumber();
498:M > }
// [...]
519:M > if ("".equals(this.destMobile)) {
520:M >  getDestMobile();
521:M > }
522:M > sendSMS(this.destMobile, "imei:" + this.imei)
```

The malicious version of the app `com.happymaau.MathRef` also changes the Manifest file to require additional permissions as well as change the behavior of the app (with malicious code). All sandboxes were able to detect this malware. In this case, the malicious version of the app changes the Android Manifest file, requiring permissions to access the network and WiFi states (Listing 4.3). These changes allow an app to view the status of all networks and make changes to configured WiFi networks.

Listing 4.3: Diffs in the `com.happymaau.MathRef` AndroidManifest file of the malicious version.

```
165:M >    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
166:M >    <uses-permission android:name="android.permission.ACCESS_WIFI_STATE"/>
```

The malicious version also introduces a method `a`, that actually collects network and WiFi information, like Mac address and the network state (see Listing 4.4). This information is then shared using an HTTP request.

Listing 4.4: Diffs in the malicious version of the class `com.mn.vymq.b.d` (app `com.happymaau.MathRef`)

```
105:M > private String a(Context paramContext)
106:M > {
107:M > String str = ((TelephonyManager)paramContext.getSystemService("phone")).getDeviceId();
108:M > StringBuilder localStringBuilder = new StringBuilder();
109:M > localStringBuilder.append(str);
110:M > paramContext = (WifiManager)paramContext.getSystemService("wifi");
111:M > if (paramContext == null) {}
112:M >  for (paramContext = null;; paramContext = paramContext.getConnectionInfo())
113:M >  {
114:M >   if (paramContext != null)
115:M >    {
116:M >      paramContext = paramContext.getMacAddress();
117:M >      if (paramContext != null) {
118:M >       localStringBuilder.append(paramContext);
119:M >      }
120:M >    }
121:M >    return a(localStringBuilder.toString());
122:M >  }
123:M > }
```

All resulting sandboxes also detected the malicious version of the app `ru.qixi.android.smartrabbi`
This particular malware also changes the Android Manifest file, requesting permission to access the location service (Listing 4.5). This permission allows access to location features, such as the Global Positioning System (GPS) on the phone, if it is enabled. Malicious applications can use these features to determine where the phone owner is, which is a classic and well-documented privacy threat.

Listing 4.5: Diffs in the `com.happymaau.MathRef` AndroidManifest file of the malicious version.

```
8:M >    <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION"/>
9:M >    <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
```

In addition, the malicious app clandestinely monitors the geographic location of the user and sink this information to a web server. Listing 4.6 shows how the method `c`, from the class named `q`, collects this sensitive information.

Listing 4.6: Diffs in the malicious version of the class `net.crazymedia.iad.d.q` (app `ru.qixi.android.smartrabbits`)

```
65:M > private Location c(Context paramContext)
66:M > {
67:M > try
68:M >  {
69:M >  if (Arrays.asList(paramContext.getPackageManager().getPackageInfo
              (paramContext.getPackageName(),4096).requestedPermissions).contains
              ("android.permission.ACCESS_FINE_LOCATION"))

70:M >   {
71:M >    paramContext = (LocationManager)paramContext.getSystemService("location");
72:M >    Criteria localCriteria = new Criteria();
73:M >    localCriteria.setAccuracy(1);
74:M >    localCriteria.setAltitudeRequired(false);
75:M >    localCriteria.setBearingRequired(false);
76:M >    localCriteria.setCostAllowed(true);
77:M >    localCriteria.setPowerRequirement(1);
78:M >    paramContext = paramContext.getLastKnownLocation
                  (paramContext.getBestProvider(localCriteria, true));

79:M >    return paramContext;
80:M >   }
81:M >  }
82:M >  catch (PackageManager.NameNotFoundException paramContext)
83:M >  {
84:M >   paramContext.printStackTrace();
85:M >   return null;
86:M >  }
87:M >  catch (Exception paramContext)
88:M >  {
89:M >   paramContext.printStackTrace();
90:M >  }
91:M >  return null;
92:M > }
```

This pattern of changing the Android Manifest file and including new method calls characterizes the classes of malwares for which the mining sandbox approach excels. In a different vein, the malicious version of the app `com.andoop.flyracing` is among the apps that none of the sandboxes could detect. Indeed, the malicious version only changes the Android Manifest file, modifying the meta-data `ADMOB_PUBLISHER_ID`. The AdMob is a monetizing service provided by Google, and changing the AdMob *publisher identifier* account redirects the advertisement's revenue to another destination. Based on this observation, we envision integrating a different approach that reasons about modifications to the Android Manifest file and that might complement the mining sandbox approach into the task for detecting malwares; since the mining sandbox approach is not able to detect malicious packages that do not introduce new method calls for sensitive APIs.

Listing 4.7: Diff in the file `com.andoop.flyracing` AndroidManifest file of the malicious version. `B` stands for the benign version, while `M` stands for the malicious version.

```
1:B < <meta-data android:name="ADMOB_\PUBLISHER_\ID"
                 android:value="a14cf7346295891"/>
---
1:M > <meta-data android:name="ADMOB_\PUBLISHER_\ID"
                 android:value="a14f099bfbf3c61"/>
```

4.3.2 Results of the second study: Use of Taint Analysis for Malware Identification

In this second study we used a taint analysis approach to mine differences between the benign and malicious versions of the 96 Android apps in our dataset. To this end we leverage the FlowDroid tool, which tracks how sensitive information flows through the apps using taint analysis algorithms. Regarding accuracy, the taint analysis approach detected 58 out of the 96 pairs in our dataset $(60, 42\%)$. That is, using the taint analysis implementation of FlowDroid alone outperforms the Monkey, DroidMate, and Humanoid sandboxes computed in the second execution (without the DroidFax static analysis algorithms). This result shows that static analysis algorithms are promising to complement the mining sandbox approach.

Finding 3 *The performance of FlowDroid to identify malicious behavior is equivalent to the performance of the mining sandbox approach supported by dynamic analysis only—i.e., without the DroidFax static analysis algorithms.*

Additionally, we investigate if we could benefit from combining the static analysis strategies from FlowDroid and DroidFax. Figure 4.5 shows a Venn-diagram summarizing the results. So, when combining the results from FlowDroid and DroidFax, we were able to detect 67 of the malicious apps (69.79%), a result compatible to the performance we found as response to the first execution of the test case generation tools—which also considers the DroidFax static analysis algorithms. More interesting, from those 67 malicious apps identified, 33 malwares had been found by both FlowDroid and DroidFax, even though they follow a completely different static analysis approach. Furthermore, FlowDroid shows to be more effective than DroidFax alone, detecting 25 malicious apps that had not been detected by DroidFax (while DroidFax detected 9 malicious apps that had not been detected by FlowDroid). The results of a logistic regression analysis, considering the model $Malware \sim Tool$, where Malware is a response variable indicating if the malware has been detected or not and Tool is either FlowDroid or the sandbox DroidFax static analysis component generates, reveals the existence of a significant difference between the performance of both tools (see Table 4.3).

Table 4.3: Results of the Logistic Regression (second study)

	Estimate	*p-value*	*C.I.*
Tool [FlowDroid]	0.4229	0.0428 **	(0.080,0.766)
Tool [DroidFax static analysis component]	-1.2730	0.0000 ***	(-1.560,-0.986)
AIC	334.61		
Num. obs.	288		

*** *p-value* < 0.001; *** *p-value* < 0.05

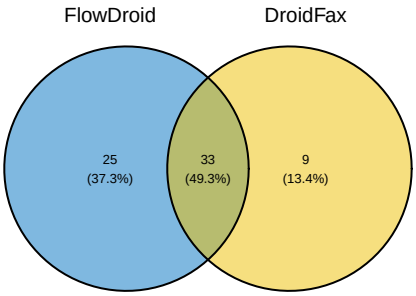


Figure 4.5: Venn Diagram highlighting the possible benefits of integrating FlowDroid and DroidFax.

Finding 4 *Integrating the results of static analysis tools (such as FlowDroid and DroidFax) seems promising, leading to a performance similar to that achieved when combining test case generation tools with the DroidFax static analysis algorithms.*

The execution of FlowDroid is also feasible: the analysis takes only 32.08 seconds per app on average, totaling a processing time of 52 minutes to analyze all 96 pairs of Android apps. Even though the time to execute the FlowDroid analysis depends on the size of the app, the longest run took only 437 seconds. Figure 4.6 summarizes the FlowDroid execution time—which most often concludes the execution in less than 50 seconds (32.11 seconds on average, with a standard deviation of 70.04).

Finally, we highlight that FlowDroid was able to detect 4 malwares among the 18 malicious Android apps that had not been detected by the sandboxes constructed in the first study. Among these four malwares, 2 are *trojans*, 1 is an *exploit*, and 1 is an *adware*.

Finding 5 *Although FlowDroid presents a performance similar to that of using the dynamic analysis approach for mining sandboxes, it was able to detect four additional malwares (out of the 18) that had not been detected in the first study.*

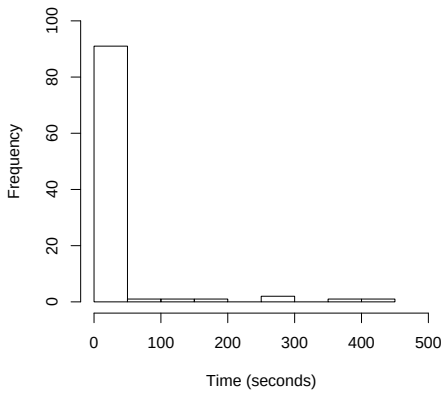


Figure 4.6: Histogram summarizing the time to execute FlowDroid

4.4 Implications

The results discussed so far bring evidence that the BLL-Study study overestimated the performance of the dynamic analysis tools in malware identification using the mining sandboxes. This finding has implications for both researchers and practitioners. First, we revisit the literature showing that DroidFax alone is also effective for mining sandboxes, being able to identify 43.75% of the malwares in our dataset. Moreover, DroidFax identifies malwares that none of the generated sandboxes were able to find, increasing the performance of the sandbox in at most 51.79% (in the case of Humanoid).

Table 4.1 in the previous section summarizes this finding: when executing the MAS approach without the support of DroidFax static analysis, Humanoid's sandbox could identify only 27 malwares (28.12% of the malwares in our dataset). Conversely, the DroidBot sandbox achieved the best performance in terms of the number of detected malware without the DroidFax support for static analysis, being able to identify 63.54% of the malwares. The message here is that researchers and practitioners should explore the use of DroidFax (or a similar tool) in conjunction with dynamic analysis techniques for mining sandboxes— reviewing the findings of the BLL-Study [3] and enriching the discussion about the limitations of static analysis for mining sandboxes [13].

In the second study we used FlowDroid to explore a novel approach for malware identification, which aims to compare the source-sink paths of two versions of an app (one known to be secure and another that might have been repackage or that might have an injected malicious behavior). Contrasting with the static analysis limitations discussed in [13], our findings indicate that this approach is also effective for malware identification. Indeed, our taint analysis approach using FlowDroid detects several malwares that none of the sandboxes generated with the dynamic analysis tools (plus the DroidFax static

analysis component) could identify (see Table 4.4). These result has also implications for both academia and industry. First, this it reinforces the benefits of integrating both static and dynamic analysis for malware identification. Second, this finding suggests that practitioners can benefit from using an integrated approach that combines the mining sandbox approach with taint analysis for malware identification.

Test Generation Tool	FlowDroid Increase	Total	%
DroidBot	6	79	82.29
Monkey	7	78	81.25
DroidMate	7	75	78.12
Humanoid	16	72	75.00
Joker	25	67	69.79

Table 4.4: Malwares detected in 96 pair (B/M) increased by the taint analysis approach

4.5 Threats to Validity

As any empirical work, this work also has limitations and threats to its validity. We organize this section using the taxonomy of Wohlin et al. [124, Chapter 8].

Conclusion Validity is concerned with the issues that might compromise the correct conclusion about the causal relation between the treatment and the outputs of an experiment. The use of inadequate statistical methods and low statistical significance are examples of threats to the conclusion validity. Besides using descriptive statistics and plots, we also leverage binomial logistic regression to support our conclusions in our two empirical studies. Indeed, the results of our logistic regression analysis give evidence about the existence of a true pattern in the data, indicating that the DroidFax static analysis component increases the performance of the sandboxes we built from the execution of the dynamic analysis tools (first study) and that FlowDroid outperforms the DroidFax static analysis component in the task of identifying malwares (second study).

Internal Validity relates to external factors that might impact the independent variables without the researchers' knowledge. Our two empirical studies are *technology-oriented* [124, 125], which are not subject to learning effect threats. Nonetheless, due to the random behavior of the test case generation tools, we should not validate the results of this experiment without considering the presence of random events in the execution. To mitigate this threat, we have used a configuration of DroidXP that runs multiple times each tool and computes the average result from those executions. So, we could adequately compare the results of our experiment with the results of the BLL-Study. Beyond that, we

tested only 96 of the original 102 pairs of apps in this experiment because the we could not execute those six pairs of apps due to crashes in the Android emulator. However, our goal here is not to conduct an exact replication of the previous work, but actually to better understand how static analysis supports and complements the mining sandbox approach for malware identification.

Construct Validity concerns possible issues that might prevent a researcher to draw a conclusion from the experimental results. The design of our first study involves one treatment (a two-level factor indicating the use or not of the DroidFax static analysis component) and three independent variables: **app id** (96 level factor), the test case generation tool (4-level factor, including DroidBot, DroidMate, Monkey, and Humanoid), and the 3-level factor repetition (we executed every tool three times for all apps, with and without the DroidFax static analysis component). The dependent variable indicates if a malware has been identified by the sandbox of a given test case generation tool built with (or without) the DroidFax static analysis component (in a particular repetition). This design leads to a total of 2304 observations, which is in conformance with the recommendations of Arcuri and Briand [126] for this kind of experiment. Our second study presents a more straightforward design, comprising a two factor treatment (FlowDroid x the DroidFax static analysis) and the same set of 96 apps of the first study. The dependent variable indicates if a malware has been identified by FlowDroid or by the sandbox the DroidFax static analysis component generates. This design leads to a smaller number of runs, but we still believe that it is sufficient to draw our conclusions (as the results of the logistic regression indicate).

External Validity concerns whether or not the researchers can generalize the results for different scenarios. Our study shares some of the threats the BLL-Study had presented. In particular, here we used the same set of pairs of apps from a *piggy-backed* dataset released by Li et al. [127]. That is, using this dataset, we could not cover all categories of Android malware. Besides that, we only used a small number of four test case generation tools in this study. To mitigate these threats and enrich the generalization of our research, we make available DroidXP, which does allow future experiments to evaluate other test case generation tools in different malware datasets.

4.6 Conclusions

In this chapter we reported the results of two empirical studies that explore techniques for Android malware identification. The first study is a non-exact replication of a previous research work [3], which investigates the Android mining sandbox approach for malware identification. There, Bao et al. report that more than 70% of the malwares in their

dataset can be detected by the sandboxes built from the execution of five test case generation tools (such as Monkey and DroidMate). Our replication study revealed that this performance is only achieved if we enable a static analysis component from DroidFax that was supposed to only instrument the Android `apk` files, though that independently contributes to building the sandboxes statically. As such, the use of DroidFax leads to an overestimation of the performance of the mining sandbox approach supported by dynamic analysis. Indeed, the execution of DroidFax alone enabled us to generate a sandbox that can identify 43.75% of the malwares from their dataset.

In the second study we investigated a new approach based on taint analysis for malware identification, which leads to promising results. First, the taint based static analysis approach detected 60.42% of the malwares in the dataset. When combining taint analysis with the mining sandbox approach, we were able to identify 82.29% of the malwares in the dataset. These results have implications for both researchers and practitioners. First, we review the literature showing, for the first time, empirical evidence that the mining sandbox approach benefits from using both dynamic and static analysis. Second, practitioners can improve malware identification using a combination of the mining sandbox approach with taint analysis. Nonetheless, both the mining sandbox approach and taint analysis present limitations. In particular, we are not able to identify a malware that uses the same set of calls to sensitive APIs of the benign version of an app, using the mining sandbox approach. Similarly, we are not able to identify a malware that presents the same paths from sources to sinks of the corresponding benign version of an app, using the taint analysis approach. To mitigate these limitations, we envision the use of other approaches—such as machine learning algorithms to classify changes in non-code assets (e.g., Android manifest files) and symbolic execution to differentiate malicious calls or source-sink paths.

Chapter 5

Assessing the MAS approach at Scale

Revisiting the Mining Android Sandbox Approach at Scale for Malware Classification

This chapter corresponds to our paper published in ECOOP 2025,
article No. 40; pp. 40:1–40:26.
doi: https://doi.org/10.4230/LIPIcs.ECOOP.2025.40.
(F. H. da Costa et al.)

5.1 Introduction

Mobile technologies, such as smartphones and tablets, have become fundamental to how we function as a society. Almost two-thirds of the world population uses mobile technologies [116, 128], with the Android Platform dominating this market and accounting for more than 70% of the *mobile market share*, with almost 2.5 million Android applications [1] (apps) available on the Google Play Store, in June 2023 [129]. As popularity rises, so does the risk of potential attacks, prompting collaborative efforts from both academia and industry to design and develop new techniques for identifying malicious behavior or vulnerable code in Android apps [130]. One popular class of Android malware is based on repackaging [3, 21], where a benign version of an app is infected with malicious code, e.g., to broadcast sensitive information to a private server [10], and subsequently shared with users using even official app stores.

[1] In this chapter, we will use the terms Android Applications, Android Apps, and Apps interchangeably, to refer to Android software applications

The Mining Android Sandbox approach (MAS approach) was initially designed to construct sandboxes based on exploratory calls to sensitive APIs [13]. The MAS approach operates in two distinct phases. In the first phase (exploratory phase), automated test case generation tools are utilized to abstract the behavior of an app, focusing on recording calls to sensitive APIs (Application Programming Interfaces). Subsequently, during normal app execution (exploitation phase), the generated sandbox blocks any calls to sensitive APIs that were not observed during the exploratory phase. Prior studies [3, 26] have investigated the effectiveness of the MAS approach in detecting potential malicious behavior in repackaged apps. These studies have also conducted comparisons of the approach's performance by employing different test case generation tools during the exploratory phase, including Monkey [17], DroidBot [20], and Droidmate [131]—bringing evidence that DroidBot outperforms other test generation tools, uncovering many potential malicious behaviors.

Nonetheless, these previous studies have two main limitations. First, they use a small dataset of malware comprising only 102 pairs of original/repackaged versions of an app—which might compromise external validity. Second, their assessments do not investigate the impact of relevant features of the repackaged apps on the accuracy of the MAS approach for malware classification, including (a) whether or not the repackaged version is a malware, (b) the similarity between the original and the repackaged versions of an app, and (c) the malware family [2] when the repackaged version of an app is a malware. These limitations compromise a broader understanding of the MAS approach performance. We present more details about the MAS approach and related work in Chapter 2.

To better understand the impact of these issues on previously published results, this chapter presents a replication of the study conducted by Bao et al. [3]. We aim to verify the original study's findings by executing the test case generation tool DroidBot [20] in the same settings as the original research. Unlike the original study, here we use a curated dataset of app pairs (original/repackaged versions) significantly larger than the dataset used in Bao et al.'s study. Our new dataset contains 4,076 pairs of original and repackaged apps. We present more details about the datasets, data collection, and analysis procedures in Section 5.2.

Negative result. Our study reveals a significantly lower accuracy (F1-score of 0.54) of the MAS approach in comparison to what the MAS approach approach performs in the small dataset (F1-score of 0.90). Since an accuracy of 0.54 is unsatisfactory for a trustworthy malware classification technique, we conduct a set of experiments to understand the reasons for the lower accuracy in our dataset. Our further assessments reveal that

[2]Malware families (such as *gappusin*, *kuguo*, *dowgin*, etc.) are often used to classify malware in groups that share similar codebases, attack methods, and objectives [54].

the MAS approach fails to correctly classify most samples from a specific set of malware families, particularly those from the *gappusin* family (a particular adware class frequently appearing in repackaged apps). Out of the total of 1,337 samples within this family in our large dataset, the MAS approach failed to classify 1,170 samples as malware correctly. Accordingly, these families are responsible for substantially reducing the recall of the MAS approach. We detail the results of our experiments in Section 6.4. We also discuss the implications and possible threats to the validity of our study in Section 6.5 and present some final remarks in Section 5.5. The main artifacts we produced during this research are available in the repository.

`https://github.com/droidxp/paper-ecoop-results`

5.2 Experimental Setup

This research aims to develop a deeper understanding of the performance of the MAS approach for detecting malware. To this end, in this chapter, we **replicate** the study by Bao et al. [3], which advocates for using the MAS approach for malware classification. However, in contrast to the original study [3], we use a dataset of repackaged apps that is an order of magnitude larger. Accordingly, we investigate the following research questions:

(RQ1) What is the impact of considering a larger and diverse dataset on the accuracy of the MAS approach for malware classification ?Answering this question may help shed light on potential generalization issues in previous studies that empirically assess the MAS approach approach to malware classification.

(RQ2) What is the influence of the similarity between the original and repackaged versions of the apps on the performance of the MAS approach for malware classification?Answering this research question helps clarify whether the similarity between an original app and its repackaged version affects the MAS approach approach's performance in malware classification.

(RQ3) What is the influence of the malware family (e.g., *gappusin*, *kuguo*, *dowgin*) on the performance of the MAS approach for malware classification ?Answering this research question may help identify potential blind spots in the MAS approach approach to malware classification, revealing possible extensions that could improve the detection of specific malware families.

In this section, we describe our study settings. First, we present our procedures to create our datasets (Section 5.2.1). Then, we describe the data collection and data analysis procedures (Sections 5.2.2 and 5.2.3).

5.2.1 Malware Dataset

To address our research questions, we contribute a dataset designed to meet two primary requirements. First, it should provide a comprehensive and up-to-date selection of Android repackaged apps. By "comprehensive", we mean at least an order of magnitude larger than the dataset used in the original study [3]. Given its comprehensiveness, we expect it to include a diverse range of malware families to ensure representativeness. Second, our dataset should be properly labeled, ensuring each sample includes key attributes such as similarity and malware family. This is particularly necessary to answer research questions RQ2 and RQ3.

Procedures for Building the Dataset

We curate our dataset in three main phases. In the first phase, we use two repositories of repackaged Android apps (RePack [10] and AndroMalPack [132]) to build the dataset we use in our research. RePack was curated using automatic procedures that extract repackaged apps from the Androzoo repository [113]. It comprises 18,073 apps, from which 2,776 are original versions of an app and the remaining ones are repackaged. RePack contains 15,297 pairs of original and repackaged Android apps, many repackaged versions of the same original app may coexist within the RePack dataset—note that all repackaged variants of a given app are derived from the same original version, as confirmed by their matching hash identifier. RePack is the leading dataset used in Android repackaged research [41], even though it only contains packages built until 2018. For this reason, we decided to include samples from the AndroMalPack dataset collected after 2018 in our research. Unlike RePack, AndroMalPack lacks information about the original apps, leading us to follow an existing heuristic [10] to identify the original versions of its repackaged apps, leading to a sample from the AndroMalPack dataset that contains 1,190 pairs (original/repackaged) of apps, all pairs satisfying our constraint of being collected after 2018. Altogether, our initial dataset contains a total of 16,487 pairs of apps.

In the second phase, we discarded some samples from our initial dataset because, during the execution of our experiments, we encountered recurrent issues related to the instrumentation of the apps using DroidFax [23]. Other problems occurred after the execution of the apps in the Android emulator, while analyzing the apps or their execution logs. More precisely, we encountered failures while instrumenting 919 original apps from our initial dataset, including both RePack and AndroMalPack. After removing these original apps from our dataset, we were left with 5,875 pairs (original/repackaged) of apps. Among these pairs, 430 repackaged apps could not be instrumented. Failures also occurred while analyzing either the original or repackaged version of 586 apps, resulting in a dataset

containing 4,742 pairs. Failures at this phase were expected, as some malware samples employ evasion tactics, such as deliberately crashing test apps in simulated environments, to avoid detection [133]. Finally, we could not install five apps in the version of the Android emulator (API level 28) we used in our research. Compared to our experience building our dataset, a more significant percentage of failures has been reported in previous research [3]. Note that we did not apply any filters to increase the representation of certain malware families in our dataset.

Third, we queried the `VirusTotal` repository to identify original versions of apps labeled as malware. Samples with such labels were excluded from our dataset, as the MAS approach assumes that the original version of an app is not malware (otherwise, the repackaged versions might also exhibit malicious behavior). `VirusTotal` is a widely recognized tool that scans software assets, including Android apps, using over 60 antivirus engines [41]. Thus, we excluded 661 samples from our dataset that do not satisfy this constraint.

In the end, we are left with our final dataset (hereafter `LargeDS`) of 4,076 apps which we use in our study. To bring evidence that we were able to reproduce the results of previous research, we also consider in our research a small dataset (`SmallDS`) used in the original study [3]. This is the same dataset referenced in Section 2.3.3 as (`LargeE`).

Features of the Datasets

We queried the `VirusTotal` repository to find out which repackaged apps in our dataset have indeed been labeled as a malware. According to `VirusTotal`, in the `SmallDS` (102 pairs), 69 of the repackaged apps (67.64%) were identified as malware by at least two security engines. Here, we consider a repackaged version of an app to be malware only if `VirusTotal` reports that at least two security engines identify malicious behavior within the asset. Although this decision aligns with previous research [134, 41], we assess its potential impact on our findings in Section 6.5. Considering the `LargeDS`, at least two security engines identified 2,895 out of the 4,076 repackaged apps as malware (71.02%). Again, in Section 6.5, we show that our results remain consistent across three additional scenarios: classifying a repackaged version of an app as malware if at least one, five, or ten `VirusTotal` engines flag it as malicious.

Classifying malware into different categories is a common practice. For instance, Android malware can be classified into categories like riskware, trojan, adware, etc. Each category might be further specialized in several malware families, depending on its characteristics and attack strategy—e.g., steal network info (IP, DNS, WiFi), collect phone info, collect user contacts, send/receive SMS, and so on [135]. According to the `avclass2` tool [136], the malware samples in the `SmallDS` come from 17 different families—most

of them from the Kuguo (49.27%) and Dowgin (17.39%) families. Our `LargeDS`, besides comprising a large sample of repackaged apps (4,076 in total), contains 116 malware families—most of them from the Gappusin (46.18%) family. Despite being flagged as malicious by at least two security engines, unfortunately `avclass2` tool cannot correctly identify the family of 253 samples in our `LargeDS`.

We also characterize our dataset according to the similarity between the original and repackaged versions of the apps, using the SimiDroid tool [137]. SimiDroid quantifies the similarity based on (a) the methods that are either identical or similar in both versions of the apps (original and repackaged versions), (b) methods that only appear in the repackaged version of the apps (new methods), and (c) methods that only appear in the original version of the apps (deleted methods). Our `LargeDS` has an average similarity score of 90.39%, with the following distribution: 87 app pairs have a similarity score below 25%, 49 pairs fall between 25% and 50%, 353 apps between 50% and 75%, and 3,587 apps exceed 75%. The `SmallDS` has an average similarity score of 89.41%.

After executing our experiments, we identified the most frequently abused sensitive APIs called by the repackaged version of our samples. We observed that upon execution of all samples from our dataset (`SmallDS` and `LargeDS`), malicious app versions injected 133 distinct methods from sensitive APIs (according to the AppGuard [120] security framework). Malicious code often exploits these APIs to compromise system security and access sensitive data. Table 5.1 lists the 10 most frequently called methods from sensitive APIs that appear only in the repackaged versions of the apps.

We must highlight that the `LargeDS` samples come from different Android app stores. Most of our repackaged apps come from a non-official Android app store, Anzhi [138]. However, some repackaged apps also come from the official Android app store, Google Play.

5.2.2 Data Collection Procedures

We take advantage of the DroidXP infrastructure [24] for data collection. DroidXP allows researchers to compare test case generation tools for malicious app behavior identification, using the MAS approach. Although the comparison of test case generation tools is not the goal of this chapter, DroidXP was still useful for automating the following steps of our study.

(Step1) **Instrumentation**: In the first step, we configure DroidXP to instrument all pairs of apps in our datasets (`SmallDS` and `LargeDS`). Here, we instrument both versions of the apps (as APK files) to collect relevant information during their execution. Under the hood, DroidXP leverages DroidFax to instrument the apps

Table 5.1: Sensitive APIs that frequently appear in the repackaged versions of the apps. The *Occurrences* column gives the number of distinct repackaged apps that introduce a call to a sensitive method.

Method of Sensitive API	Occurrences
android.telephony.TelephonyManager: int getPhoneType()	311
android.telephony.TelephonyManager: java.lang.String getNetworkOperatorName()	297
android.location.LocationManager: java.lang.String getBestProvider(android.location.Criteria,boolean)	292
android.telephony.TelephonyManager: int getSimState()	284
java.lang.reflect.Field: java.lang.Object get(java.lang.Object)	277
android.net.NetworkInfo: java.lang.String getTypeName()	271
android.database.sqlite.SQLiteDatabase: android.database.Cursor query(java.lang.String,java.lang.String[],...,...,...,...,...)	270
java.lang.reflect.Field: int getInt(java.lang.Object)	250
android.net.wifi.WifiInfo: java.lang.String getMacAddress()	238
android.telephony.TelephonyManager: java.lang.String getNetworkOperator()	237

and collect static information about them. To improve the performance across multiple executions, this phase executes only once for each version of the apps in our dataset.

(Step2) **Execution**: In this step, DroidXP first installs the (instrumented) version of the APK files in the Android emulator we use in our experiment (API 28) and then starts a test case generation tool for executing both app versions (original and repackaged). We execute the apps via DroidBot [20], mainly because the original research we replicate here reports that DroidBot leads to the best accuracy of the MAS approach for malware identification. Since previous studies suggest that DroidBot's coverage nearly reaches its maximum within one minute [3], we run each app for three minutes. To mitigate the randomness inherent in test case generation tools, we repeat this process three times. To also ensure that each execution gets the benefit of running on a fresh Android instance without biases that could stem out of history, DroidXP wipes out all data stored on the emulator that has been collected from previous executions.

(Step3) **Data Collection**: After the execution of the instrumented apps, once again, DroidXP leverages DroidFax, this time to collect all relevant information (such as calls to sensitive APIs, test coverage metrics, and so on). We use this information to analyze the performance of the MAS approach for detecting malicious behavior.

5.2.3 Data Analysis Procedures

We consider that the MAS approach builds a sandbox that labels a repackaged version of an app as malware if there is at least one call to a sensitive API that (a) was observed while executing the repackaged version of the app and that (b) was not observed while executing the original version of the same app. If the set of sensitive methods that only the repackaged version of an app calls is empty, we conclude that the sandbox does not label the repackaged version of an app as malware. The set of sensitive APIs we use was defined in the AppGuard framework [120], which was based on the mapping from sensitive APIs to permissions proposed by Song et al. [29]. We triangulate the results of the MAS approach classification with the outputs of `VirusTotal`, which might lead to one of the following situations:

- **True Positive (TP)**. The MAS approach labels a repackaged version as malware and, according to `VirusTotal`, at least two security engines label the asset as a malware.

- **True Negative (TN)**. The MAS approach does not label a repackaged version as malware and, according to `VirusTotal`, at most one security engine labels the asset as a malware.
- **False Positive (FP)**. The MAS approach labels a repackaged version as malware and, according to `VirusTotal`, at most one security engine labels the asset as a malware.
- **False Negative (FN)**. The MAS approach does not label a repackaged version as malware, and according to `VirusTotal`, at least two security engines label the asset as a malware.

In Section 6.4 we compute *Precision*, *Recall*, and *F-measure* (F_1) from the number of true-positives, false-positives, and false-negatives (using standard formulae). We use basic statistics (average, median, standard deviation) to identify the accuracy of the MAS approach for malware classification, using both datasets—i.e., the `SmallDS` with 102 pairs of apps and `LargeDS` with 4,076 pairs. We use the Spearman Correlation [139] method and Logistic Regression [121] to understand the strengths of the associations between the similarity index between the original and the repackaged versions of a malware with the MAS approach accuracy—that is, if the approach was able to classify an asset as malware correctly. We also use existing tools to reverse engineer a sample of repackaged apps in order to better understand the (lack of) accuracy of the MAS approach.

Table 5.2 highlights the differences between the original study [3] and our replication study. In the best-case scenario, where no re-executions are required, our experiment would take at least 611 hours. In contrast, under the best conditions, the original experiment's execution would last 60 hours. This difference is one of the reasons we focus our research on DroidBot, the test case generation tool that demonstrated the best performance in the original study.

Study Feature	**Original Study**	**Replication Study**
Dataset	102 pairs of samples	4,076 pairs of samples
Execution time	One minute	Three minutes
Number of executions	Single execution	Three executions
Metrics	Malware prevalence	Precision, Recall, and F1-score
Test case generation tool	Six different tools (DroidBot with best performance)	DroidBot only

Table 5.2: Characterization of this replication study

5.3 Results

In this section, we detail the findings of our study. We remind the reader that this replication study's primary goal is to better understand the strengths and limitations of the MAS approach for malware detection by replicating the work of Bao et al., using DroidBot as the test case generation tool. We explore the results of our research using two datasets: the `SmallDS` (102 app pairs), and `LargeDS` (4,076 pairs).

5.3.1 Exploratory Data Analysis of Accuracy

`SmallDS`. Considering the `SmallDS` (102 apps), the MAS approach for malware detection classifies a total of 69 repackaged versions as malware (67.64%). This result is close to what Bao et al. reported [3]. That is, in their original paper, the MAS approach using DroidBot classifies 66.66% of the repackaged version of the apps as malware [3]. This result confirms that we could reproduce the findings of the original study using our implementation settings of the MAS approach.

Finding 1. We were able to reproduce the results of existing research using our implementation of the MAS approach, achieving a malware classification in the `SmallDS` close to what has been reported in previous studies.

In the original study [3], the authors assume that all repackaged versions are malware and contain a malicious code. For this reason, the authors do not explore accuracy metrics (such as Precision, Recall, and F-measure (F_1))—all repackaged apps labeled as malware are considered true positives in the original study. As we mentioned, in this chapter we take advantage of `VirusTotal` to label our dataset and build a ground truth: In our datasets, we classify a repackaged version of an app as malware if, according to our `VirusTotal` query results, at least two security engines identify malicious behavior in the asset. This decision aligns with existing recommendations [134, 41]). The first row of Table 5.3 shows that the MAS approach achieves an accuracy of 0.90 when considering the `SmallDS`. Nonetheless, the MAS approach fails to classify seven assets as malware on the `SmallDS` correctly (FN column, first row of Table 5.3), and wrongly labeled the repackaged version of six apps as malware (FP column).

Table 5.3: Accuracy of the MAS approach in both datasets.

Dataset	TP	FP	FN	Precision	Recall	F_1
`SmallDS` (102)	63	6	7	0.91	0.90	0.90
`LargeDS` (4,076)	1,175	220	1,720	0.84	0.40	0.54

LargeDS. Surprisingly, considering our complete dataset (4,076 apps), the MAS approach labels a total of 1,395 repackaged apps as malware (34.22% of the total number of repackaged apps)—for which the repackaged version calls at least one additional sensitive API. Our analysis also reveals a **negative result** related to the accuracy of the approach: here, the accuracy is much lower in comparison to what we reported for the SmallDS (see the second row of Table 5.3): F_1 dropping from 0.90 to 0.54. This result indicates that, when considering a large dataset, the accuracy of the MAS approach using DroidBot drops significantly.

Finding 2. The MAS approach for malware detection leads to a substantially lower performance on the LargeDS (4,076 pairs of apps), dropping F1-score from 0.90 to 0.54 in comparison to what we observed in the SmallDS.

Therefore, the resulting sandbox we generate using DroidBot suffers from a significantly low accuracy rate when considering a large dataset. This is shown in the second row of Table 5.3. The negative performance of the MAS approach in the LargeDS encouraged us to endorse efforts to identify potential reasons for this phenomenon and motivated us to explore the research questions RQ2 and RQ3.

5.3.2 Assessment Based on Similarity Score

Figure 5.1 shows the Similarity Score distribution over the LargeDS we use in our research. Recall that the Similarity Score measures how similar an app's original and repackaged versions are. The complete dataset averages a Similarity Score of 0.90 (with a median of 0.98 and standard deviation of 0.18).

In this section we investigate how the Similarity Score influences the accuracy of the MAS approach—which might help us understand if it might explain the low small performance of the MAS approach in the LargeDS. To this end, we leverage Logistic Regression to quantify the relationship between Similarity Score and F1-score. This analysis excludes instances of true negatives (i.e., cases where the repackaged version is benign according to VirusTotal and the MAS approach correctly labels it as benign). As such, we test the following null hypothesis:

> H_0 Similarity Score does not influence the accuracy of the MAS approach for malware detection.

The logistic regression results suggest that we should reject our null hypothesis (p-value $= 2.22 \cdot 10^{-16}$). This finding indicates that the accuracy of the MAS approach on LargeDS is influenced by the similarity between the original and repackaged versions of an app.

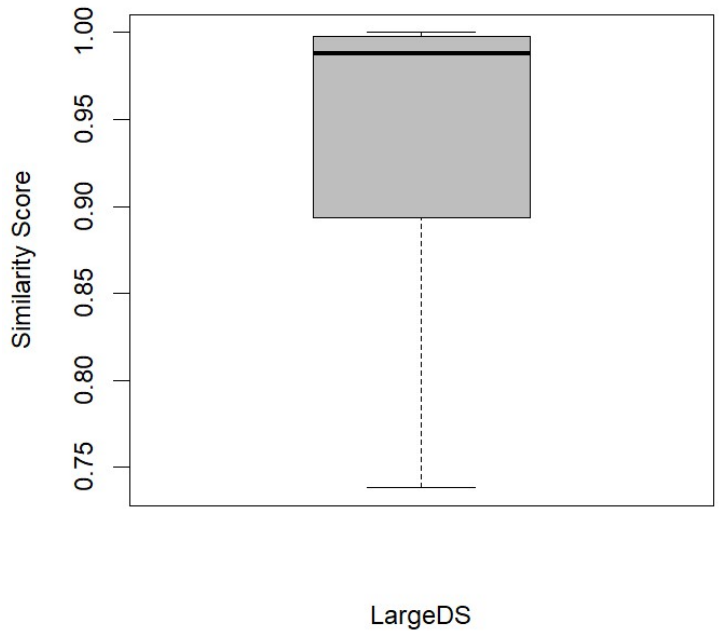


Figure 5.1: Similarity Score of the malware samples in the `LargeDS`. The boxplots in the figure do not show outliers.

Finding 3. There is an association between the Similarity Score and the MAS approach performance, which means that the similarity between the original and repackaged versions of an app can explain the performance of the MAS approach for malware classification.

To clarify the association between Similarity Score and accuracy, we use the *K-Means* algorithm to split the `LargeDS` into ten clusters—according to the Similarity Score. We then estimate the percentage of correct classifications for each cluster, as shown in Table 5.4. Note that the MAS approach achieves the highest percentage of correct classification (77.35%) for the second cluster (cId = 2), which presents an average Similarity Score of (0.56). Nonetheless, the cluster cId = 10, with a larger number of samples (1,302) and Similarity Score (0.99), presents a percentage of correct classifications of 26.5%. We can observe that as the average similarity rate decreases, there is a tendency toward greater accuracy in the MAS approach. Hence, the average similarity score could explain the poor performance of the MAS approach on the `LargeDS`, especially considering that most samples exhibit a high average similarity of 0.99.

Table 5.4: Characteristics of the clusters. Note there is a specific pattern associating the percentage of correct answers with the Similarity Score. For this analysis, we removed the true negatives in our dataset.

cId	Similarity Score	Samples	Correct Answers	%
1	0.42	42	30	71.42
2	0.56	181	140	77.35
3	0.68	131	98	74.81
4	0.80	170	104	61.18
5	0.88	263	83	31.56
6	0.91	236	129	54.66
7	0.95	167	51	30.54
8	0.97	150	67	44.67
9	0.98	421	112	26.60
10	0.99	1302	345	26.50

5.3.3 Assessment Based on Malware Family

As we discussed in the previous section, the similarity assessment partially explains the low performance of the MAS approach on the `LargeDS`. Since the `LargeDS` covers a wide range of malware families, we investigate the hypothesis that the diversity of malware families in the `LargeDS` also contributes to the poor performance of the MAS approach on the `LargeDS`. Indeed, in the `LargeDS`, we identified a total of 116 malware families, though the most frequent ones are *gappusin* (1,337 samples), *revmob* (207 samples), *dowgin* (183 samples) and *airpush* (120 samples). Together, they account for 63.79% of the repackaged apps in our `LargeDS` labeled as malware according to `VirusTotal`.

This family distribution in the `LargeDS` is different from the family distribution in the `SmallDS` (used in the original study)—where the families *kuguo* (34 samples), *dowgin* (12 samples), and *youmi* (5 samples) account for 73.91% of the families considering the 69 repackaged apps in the `SmallDS` for which `VirusTotal` labels as malware. Most important, in the `SmallDS`, there is just one sample from the *gappusin* family and no sample from *revmob* family, two of the most frequent families in our `LargeDS`. This observation leads us to the question: how does the MAS approach perform when considering only samples from the *gappusin* and *revmob* families?

The confusion matrix of Table 5.5 summarizes the accuracy assessment of the MAS approach considering only the *gappusin* and *revmob* samples in the `LargeDS`. To make clear, `VirusTotal` classifies as malware all repackaged versions in the *gappusin* and *revmob* family. It is worth noting that the MAS approach failed to classify correctly 1,170 (87.5%) samples of *gappusin* as malware. Similarly, 92 samples (44.44%) from *revmob* were not classified as malware. Furthermore, if we exclude the *gappusin* and *revmob* samples from the `LargeDS`, the recall of the MAS approach increases to 0.72, which, although improved,

remains relatively low compared to the original studies.

Table 5.5: Confusion matrix of the MAS approach when considering only the samples from the *gappusin* and *revmob* family in the `LargeDS`.

Actual Condition	Predicted Condition	
	Benign	Malware
Benign (0)	TN (0)	FP (0)
Gappusin (1,337)	FN (1,170)	TP (167)
Revmob (207)	FN (92)	TP (115)

Finding 4. The MAS approach fails to correctly identify 87.50% of the samples from the *gappusin* family and 44.44% of the samples from the *revmob* as malware. Just like the Similarity Score, the presence of some malware families with a high false negative rate also influences the low recall of the MAS approach in the `LargeDS`

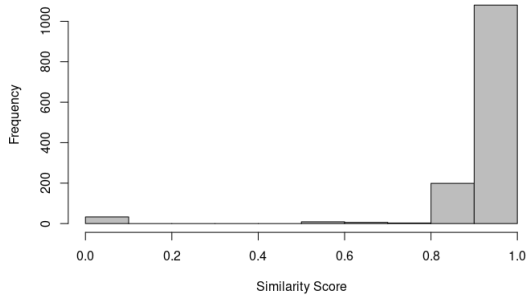
We further analyze the samples from the *gappusin* and *revmob* malware families in our dataset, given their relevance to the negative results presented in the chapter. First, we examined the Similarity Score of the samples. Figure 5.2 shows a histogram of the Similarity Score for both families. Most repackaged versions are similar to the original ones, with an average Similarity Score of 0.94, a median of 0.99, and a standard deviation (SD) of 0.16 for the *gappusin* family. For the *revmob* family, the average Similarity Score is 0.81, the median is 0.91, and the SD is 0.26.

We also reverse-engineered samples from both families. Due to the significant effort required for reverse engineering, we limited our analysis to a sample of 30 *gappusin* and 30 *revmob* malware samples, using the `SimiDroid`[3], `apktool` [4], and `smali2java` [5] tools. Considering this sample, the median Similarity Score is 0.99 and 0.90 for the *gappusin* and *revmob* families, respectively. Table 5.6 and Table 5.7 summarize the outputs of `SimiDroid` for these samples.

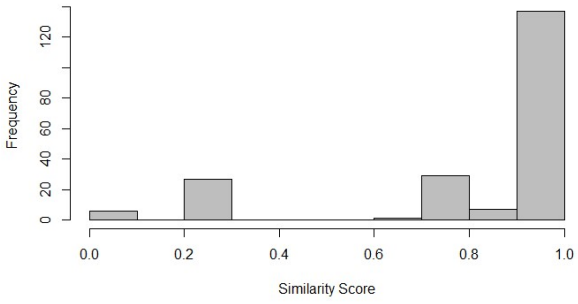
Regarding the *gappusin* malware, the similarity assessment of this sample of 30 apps reveals a few modification patterns when comparing the original and the repackaged versions. First, no instance in this *gappusin* sample dataset modifies the Android Manifest file to require additional permissions. In most cases, the repackaged version just changes the Manifest file to modify either the package name or the main activity name. Moreover, 29 out of the 30 samples in this dataset **modifies** the method `void onReceive(Context,`

[3]https://github.com/lilicoding/SimiDroid
[4]https://ibotpeaches.github.io/Apktool/
[5]https://github.com/AlexeySoshin/smali2java

`Intent)` of the class `com.games.AdReciver`. Although the results of the decompilation process are difficult to understand in full (due to code obfuscation), the goal of this modification is to change the behavior of the benign version, so that it can download a different version of the `data.apk` asset. Figure 5.3 shows the code pattern of the `onReceive` method present in the samples. This modification typically uses a new method (`public void a(Context)`) in the repackaged versions, often introduced into the same class (`AdReceiver`). Since there are no additional calls to sensitive APIs, the MAS approach fails to correctly label the *gappusin* samples. This limitation holds regardless of our experimental choices, such as using the DroidBot tool (instead of more recent test case generation tools) or running the samples for three minutes.



(a) Similarity Score for the samples in the *gappusin* family.



(b) Similarity Score for the samples in the *revmob* family.

Figure 5.2: Histogram of the Similarity Score for the samples in the *gappusin* and *revmob* families.

```
public void onReceive(Context context, Intent intent) {
  SharedPreferences sp = context.getSharedPreferences(String.valueOf("com.")+"game."+"param",0);
  int i = sp.getInt("sn", 0) + 1;
  System.out.println("sn: " + i);
  if (i < 2) {
    mo4a(context);
    SharedPreferences.Editor edit = sp.edit();
    edit.putInt("sn", i);
    edit.commit();
  } else if (!new C0004b(context).f7h.equals("")) {
    String str1 = context.getApplicationInfo().dataDir;
    String str2 = String.valueOf(str1) + "/fi" + "les/d" + "ata.a" + "pk";
    String str3 = String.valueOf(str1) + "/files";
    String str4 = String.valueOf("com.") + "ccx." + "xm." + "SDKS" + "tart";
    String str5 = String.valueOf("InitS") + "tart";
    String str6 = "ff048a5de4cc5eabec4a209293513b6e";
    C0003a.m3a(context, str2, str3, str4, str5, str6);
    SharedPreferences.Editor edit2 = sp.edit();
    edit2.putInt("sn", 0);
    edit2.commit();
  }
}
```

Figure 5.3: Method introduced in 29 out of 30 *gappusin* malware we randomly selected from the `LargeDS`.

Our assessment also reveals recurrent modification patterns that **delete** methods in the repackaged version of the apps. For instance, 20 repackaged apps in our *gappusin* sample of 30 malware remove the method `void b(Context)` from the class `com.game.a`. This class extensively uses the Android reflection API. Although it is not clear the real purpose of removing these methods, that decision simplifies the procedure of downloading a `data.apk` asset that is different from the asset available in the original version of the apps. Removing those methods might also be a strategy for antivirus evasion. For instance, although some usages of the class `DexClassLoader` might be legitimate, it allows specific types of attack based on dynamic code injection [140]. As such, antivirus might flag specific patterns using the Android reflection API suspect. Unfortunately, the MAS approach also fails to identify a malicious behavior with this type of change (i.e., changes that remove methods), again, regardless of the decisions we follow in our experiment. Listing 5.4 shows an example of code pattern frequently removed from the repackaged versions from the *gappusin* family.

```
public static void m7a(Activity activity, String str, String str2, ..., String str5) {
  try {
    Class loadClass = new DexClassLoader(..., activity.getClassLoader()).loadClass(str3);
    Object newInstance = loadClass.getConstructor(new Class[0]).newInstance(new Object[0]);
    Method method = loadClass.getMethod(str4, new Class[]{Activity.class, String.class});
    method.setAccessible(true);
    method.invoke(newInstance, new Object[]{activity, str5});
  } catch (Exception e) {
    e.printStackTrace();
  }
}
```

Figure 5.4: Example of method that is typically removed from the repackaged apps of the *gappusin* family.

In summary, our reverse engineering effort brings evidence that malware samples from the *gappusin* family neither modify the Android Manifest files nor call additional sensitive APIs. It acts as a downloader for further malicious app [54]—which reduces the ability of the MAS approach to classify a sample as a malware correctly. Both versions (original/repackaged) from the *gappusin* family have the same behavior for showing advertisements to the user, however, the repackaged version has additional call sites to the advertisement API and the advertisement sources are different.

Similar to the approach used for *gappusin* samples, we also reverse-engineered a random selection of 30 samples from the *revmob* family that were not detected by the MAS approach. As with the *gappusin* family samples, no instance from the *revmob* family modifies the Manifest file or inserts extra calls to sensitive APIs, making it harder for the MAS approach to label the samples as malware correctly. However, our reverse engineering reveals that all apps store a file with the extension ".so" (Shared Object files) in their lib directory. These files are dynamic libraries containing native code written in C or C++, and are often used by apps for performance reasons, when resource-intensive tasks need to be performed [141].

Unfortunately, creating malicious repackaged apps using ".so" files is also possible, as they can be replaced by a version containing harmful code [142]. The Shared Object files also allow for attacks based on dynamic code injection [140], considering that Android apps can use methods like `System.loadLibrary()` or `System.load()` to download malicious ".so" files from a remote server. Once on the device, malicious apps can use these files to interface with Java code in Android apps, via the Java Native Interface (JNI), performing malicious operations on low-level code and bypassing security mechanisms, like the MAS approach.

Our assessment confirms that all *revmob* samples contain Java code that loads a native library. In particular, to load these libraries, the samples use the `loadLibrary` method of the `System` class, which is called in the static constructor of the `mainActivity` class. The `loadLibrary` method takes "game" as an argument, and the code automatically searches the default lib directory for the `.so` file named (`lib+argument`). The "lib" directory contains the `libgame.so` file in all samples. Figure 5.5 presents the code pattern of the `mainActivity` class found in the samples from *revmob* family.

```
public class PZPlayer extends Cocos2dxActivity {
    // ...
    System.loadLibrary("game");
    // ...
}
```

Figure 5.5: Thie code links this java file into libgame shared library

The `libgame.so` file contains compiled code written in C or C++, which is loaded into memory and linked to the apps at runtime. Although machine code is difficult to analyze, all the files include the `JNI_OnLoad` function, which the JNI implementation automatically uses to link Java methods and native functions. When we analyzed the ".so" file, we found that they all differ in size and content between the original and repackaged apps. It is possible that changes of interest occurred in the native `libgame.so` file and have gone unnoticed by the MAS approach. Again, since the MAS approach only considers differences in calls to sensitive APIs, it is unlikely to correctly classify these samples using other test case generation tools or by extending the execution time during its exploratory phase.

5.4 Discussion

In this section, we answer our research questions, summarize the implications of our results, and discuss possible limitations of our study that might threaten the validity of the results presented so far.

5.4.1 Answers to the Research Questions

The results we presented in the previous sections allow us to answer our three research questions, as we summarize in the following.

- **Performance of the MAS approach (RQ1).** Our study indicates that the accuracy of the MAS approach reported in previous studies [3, 26] does not generalize to a larger dataset. That is, while in our reproduction study (using the `SmallDS` of previous research) the MAS approach leads to an accuracy of 0.90, we observed a drop of precision and recall that leads to an accuracy of 0.54 in the presence of our `LargeDS` (4,076 pairs of original and repackaged versions of Android apps).
- **Similarity Analysis (RQ2).** Our results bring evidence about the association between the similarity of the original and repackaged versions of an app and the ability of the MAS approach to correctly classify a repackaged version of an app as a malware. Therefore, the similarity assessment is relevant for explaining the performance of the MAS approach to classify certain repackaged versions of an app as malware.
- **Malware Family Analysis (RQ3).** The results indicate that some families are responsible for the largest number of false negatives in the complete dataset. We specifically further investigate the *gappusin* and *revmob* families—a particular type

of Adware, designed to display advertisements while an app is running automatically. After reverse engineering a sample of 60 malware apps from *gappusin* and *revmob* family, we confirmed that the MAS approach cannot identify the patterns of changes introduced in the repackaged versions of the apps. The prevalence of the *gappusin* and *revmob* families in the Android malware landscape accounts for the poor performance of the MAS approach in malware classification on the large dataset.

5.4.2 Implications

Contrasting to previous research works [3, 21, 26], our results lead to a more systematic understanding of the strengths and limitations of using the MAS approach for malware classification. In particular, this is the first study that empirically evaluates the MAS approach considering as ground truth the outcomes of `VirusTotal`—a common practice in the malware identification research. This decision allowed us to explore the MAS approach performance using well-known accuracy metrics (i.e., precision, recall, and F_1 score). Contrasting with previous studies that assume that all repackaged versions of the apps were malware. Our triangulation with `VirusTotal` reveals this is not true. Although the MAS approach presents a good accuracy for the `SmallDS` ($F_1 = 0.90$), in the presence of a large dataset the MAS approach accuracy drops significantly ($F_1 = 0.54$).

We also reveal that some families in the `LargeDS` are responsible for a large number of false negatives, compromising the accuracy of the MAS approach. Altogether, the takeaways of this research are twofold:

- Negative result: the MAS approach for malware detection exhibits a much higher false negative rate than previous research reported.
- Future directions: Researchers should advance the MAS approach for malware detection by exploring more sophisticated techniques to differentiate between benign and malicious apps. In particular, since our reverse engineering results suggest that *gappusin* and *revmob*—two recurrent malware families—use the network to download new assets, new approaches might benefit from monitoring not only calls to sensitive APIs but also network traffic, as well as mining sensitive calls to native APIs embedded in `so` files. The versatility of the Java Native Interface (JNI) has introduced challenges. Malware authors increasingly use the native layer to hide malicious code, making both static and dynamic analysis more difficult. The current state of the art in sandbox mining overlooks native calls.

5.4.3 Threats to Validity

There are some threats to the validity of our results. Regarding **external validity**, one concern relates to the representativeness of our malware datasets and how generic our findings are. Indeed, mitigating this threat was one of the motivations for our research, since, in the existing literature on the MAS approach for malware classification, researchers had explored just one dataset of 102 pairs of original/repackaged apps. Curiously, for this small dataset, the performance of the MAS approach is substantially superior to its performance on our `LargeDS` (4,076 pairs of apps).

We contacted the authors of the Bao et al. original research paper [3], asking them if they had used any additional criteria for selecting the pairs of apps in their dataset. Their answers suggest the contrary: they have not used any particular app selection process that could explain the superior performance of the MAS approach for the `SmallDS`. We believe our results in the `LargeDS` generalize better than previous research work, since we have a more comprehensive collection of malware with different families and degrees of similarity. Nonetheless, our research focuses only on Android repackaged malware. Thus, we cannot generalize our findings to malware that targets other platforms or uses different approaches to instantiate a malicious asset. Besides that, repackaging is a recurrent approach for implementing Android malware.

Regarding **conclusion validity**, during the exploratory phase of the MAS approach, we collected the set of calls to sensitive APIs the original version of an app executes, while running a test case generation tool (DroidBot). In the exploratory phase, the MAS approach assumes the existence of a benign original version of a given app. We also query `VirusTotal` to confirm this assumption, and found that the original version of seven (out 102) apps in the `SmallDS` contains malicious code. We believe the authors of previous studies carefully check that assumption, and this difference had occurred because the outputs of `VirusTotal` change over time [134], and a dataset that is consistent on a given date may not remain consistent in the future. Therefore, while reproducing this research, it is necessary to query `VirusTotal` to get the most up-to-date classification of the assets, which might lead to results that might slightly diverge from what we have reported here. Besides that, in the `LargeDS` we only consider pairs of original/repackaged apps for which `VirusTotal` classifies the original version as benign.

Regarding **construct validity**, we address the main threats to our study by using simple and well-defined metrics that are in use for this type of research: number of malware samples the MAS approach correctly/wrongly classify in a dataset (true positives/false negatives). We computed the accuracy results using precision and recall based on these metrics. In a preliminary study, we investigated whether or not the MAS approach would classify an original version of an app as malware, computing the results of the test case

generation tools in multiple runs. After combining three executions in an original version to build a sandbox, we did not find any other execution that could wrongly label an original app as malware. Also, we label a repackaged version of an app as malware only if `VirusTotal` reports that at least two engines detect suspicious behavior in that asset. This decision might be viewed as either a weak or strong constraint and could raise concerns about construct validity. However, when we relax this constraint and label an asset as malware whenever at least one engine detects suspicious behavior, precision improves to 0.85, but recall drops to 0.39. Overall, the accuracy of the MAS approach remains almost unchanged ($F_1 = 0.53$)—still significantly lower than the precision of the MAS approach for `SmallDS`. We also evaluated accuracy by considering an asset as malware when at least five or ten `VirusTotal` security engines flagged it. As shown in Table 5.8, the results did not diverge significantly from what we have reported in this chapter.

5.5 Conclusions

To better understand the strengths and limitations of the MAS approach for repackaged malware detection, this chapter reported the results of an empirical study that replicates previous research works [3, 26]. The study utilizes a more diverse dataset compared to those used in previous research, with the aim of providing a more comprehensive evaluation of the approach. To our surprise, compared to published results, the performance of the MAS approach drops significantly for our comprehensive dataset (F_1 score reduces from 0.90 in previous papers to 0.54 here). This result is partially explained by the high prevalence of specific malware families (named *gappusin* and *revmob*), whose samples are incorrectly classified by the MAS approach. We also report the results of a reverse engineering effort, whose goal was to understand the characteristics of the *gappusin* and *revmob* family that reduce the performance of the MAS approach for malware classification. Our reverse engineering effort revealed common changing patterns in the *gappusin* repackaged versions of original apps, which mostly use reflection to download an external apk asset for handling advertisements without introducing additional calls to sensitive APIs. Similarly, the *revmob* family does not include any additional calls to sensitive resources; however, it often uses JNI to interact with native code, which can be used to perform malicious operations at a low level, compromising the effectiveness of the MAS approach for malware identification. These negative results highlight the current limitations of the MAS approach for malware classification and suggest the need for further research to integrate the MAS approach with other techniques for more effective malware identification.

Table 5.6: Summary of the outputs of the `SimiDroid` tool for the sample of 30 *gappusin* malware. (IM) Identical Methods, (SM) Similar Methods, (NM) New Methods, and (DM) Deleted Methods.

Hash	Similarity Score	IM	SM	NM	DM
33896E	0.9994	3205	2	0	0
0C962D	0.9994	3413	1	1	10
BCDF91	0.9992	2645	2	0	0
01ECE4	0.9991	5697	4	1	10
A306DA	0.9989	1886	1	1	6
4010CA	0.9987	3721	1	4	6
5B5F2D	0.9983	1164	2	3	0
010C07	0.9982	2248	4	3	0
F9FC04	0.9982	1121	1	1	6
E29F53	0.9976	842	1	1	6
FE76EB	0.9976	839	1	1	6
842BD5	0.9973	2249	3	3	3
295B66	0.9972	1081	2	1	10
92209D	0.9971	698	2	3	0
0977B0	0.9969	1613	4	1	10
347FCF	0.9967	613	1	1	6
00405B	0.9965	864	2	1	10
67310E	0.9957	1164	2	3	3
CCD29E	0.9954	436	2	0	0
610113	0.9941	836	4	1	10
A871E0	0.9941	836	4	1	10
ECEA10	0.9913	229	1	1	6
E53FAA	0.9889	267	2	1	10
723C23	0.9870	228	2	1	10
D95B6E	0.9870	833	10	1	10
17722D	0.9743	265	6	1	10
537492	0.9504	134	6	1	10
078E0A	0.9504	134	6	1	10
D83F1C	0.9494	150	2	6	6
E5D716	0.8840	2035	68	199	199

Table 5.7: Summary of the outputs of the `SimiDroid` tool for the sample of 30 *revmob* malware. (IM) Identical Methods, (SM) Similar Methods, (NM) New Methods, and (DM) Deleted Methods.

Hash	Similarity Score	IM	SM	NM	DM
14BBE2	0.9940	3348	6	532	14
BFEF74	0.9940	3348	6	532	14
A3FACA	0.7918	2667	80	112	1 621
10F22D	0.9940	3348	6	532	14
50193A	0.9940	3348	6	532	14
5A7536	0.9940	3348	6	532	14
BCC0DB	0.7918	2667	80	112	1 621
E866CB	0.9940	3348	6	532	14
CDD316	0.9940	3348	6	532	14
DF39F6	0.7918	2667	80	112	1 621
3FFAFF	0.9121	3072	184	628	112
C8C63D	0.9940	3348	6	532	14
48C562	0.9121	3072	184	628	112
D27F26	0.7918	2667	80	112	1 621
F4BBEC	0.9121	3072	184	628	112
BCF14C	0.9127	3074	182	628	112
7FBF11	0.7918	2667	80	112	1 621
9D35D4	0.7918	2667	80	112	1 621
D1B27E	0.9940	3348	6	532	14
94DD4B	0.9940	3348	6	532	14
2D217E	0.7918	2667	80	1121	621
66F167	0.7918	2667	80	1121	621
155D4A	0.9940	3348	6	532	14
8CB780	0.9127	3074	82	628	112
C251FA	0.9940	3348	6	532	14
40487B	0.7918	2667	80	1121	621
F29692	0.9940	3348	6	532	14
0E3679	0.9127	3074	182	628	112
7A4F31	0.9121	3072	184	628	112
BB3EDE	0.7105	2393	256	1217	719

Table 5.8: Accuracy of the MAS approach at `LargeDS` (4,076 pairs) based on engines.

Engine(s)	TP	FP	FN	Precision	Recall	F_1
At least 01	1,222	220	1,900	0.85	0.39	0.53
At least 02	1,175	220	1,720	0.84	0.40	0.54
At least 05	1,087	220	1,578	0.83	0.40	0.54
At least 10	1,002	220	1,469	0.81	0.40	0.54

Chapter 6

DroidXPFlow

Network Flow Analysis for Android Malware Detection: Addressing Blind-spots of the MAS approach

At the time of writing this thesis, the results presented in this chapter have not been published in any venue.
(F. H. d. Costa et al)

6.1 Introduction

Android is a robust Linux-based operating system widely used in mobile technology. It has more than 2.5 million Android applications [1] (apps) available in the official Google Play Store until June 2023 [129]. As its popularity rises, so does the risk of potential attacks, making Android-based devices prime targets for malicious apps (malware). In general, the main aim of malware is to gain unauthorized access to and exploit sensitive resources on a device [143, 144]. This can lead to various risks, including disrupted device functionality, battery drain, information leakage, and other threats [143, 35].

A prevalent form of Android malware involves repackaging legitimate apps [3, 9]. These malicious variants can insert or modify the original apps with harmful code and release them on (un)official third-party markets [37]. Researchers [37, 35, 145] show that 86% of Android malicious apps are repackaged, highlighting the prevalence of this approach to inject malicious behavior. To counter this, several Android malware detection techniques have been developed. For example, the Mining Android Sandbox (hereafter MAS approach) for malware detection, adapted from [13], relies on calls to sensitive APIs

[1] In this chapter, the terms Android Applications, Android Apps, and Apps will be used interchangeably to refer to software applications for the Android platform.

to check whether a repackaged version of an app is malicious or not [3, 78]. The original MAS approach leverages static and dynamic analysis on Android apps to protect sensitive resources at a fine-grained level by limiting access to sensitive APIs.

Focused on app behavior abstraction, the MAS approach has proven effective in detecting repackaged malware, as demonstrated in previous work [3], which classified as malware 77 out of 102 app pairs (original and repackaged versions of an app) [2]. However, the Bao et al. study [3] evaluated the technique using a dataset comprising only 102 app pairs, with a limited number of malware families. Using the same dataset from Bao et al., Costa et al. [78] presented an in-depth analysis of the MAS approach that highlights the contributions of the static and dynamic analysis components to malware detection, bringing evidence that both techniques complement each other.

Francisco et al. [146] (FR-Study), presented an empirical evaluation of the MAS approach using a larger dataset (hereafter referred to as `LargeDS`), which contains 4,076 pairs of apps and 116 malware families. There, the author evidence that, when applied to the `LargeDS`, the accuracy of the MAS approach drops significantly, with an F1-score of 0.54. This suggests that the effectiveness of the MAS approach in detecting and preventing malicious behaviors may not be generalizable to larger datasets.

Motivated by the negative results reported in FR-Study, in this chapter, we propose and evaluate DroidXPflow, a new technique for Android malware identification that (a) leverages DroidXP infrastructure [24] to collect the network traffic of the apps (while they execute using a test generation tool like DroidBot [20]) and (b) benefits from machine learning (ML) algorithms to classify the repackaged version of the apps as malware / non-malware using network traffic data. We propose DroidXPflow because we do not find published tools or replication packages that explore Android malware identification using network flow data, even after contacting the authors of papers [90, 93, 94, 86, 95]. The results of the DroidXPflow evaluation show that dynamic network traffic analysis, supported by ML techniques, achieves an accuracy (F1-score) of 0.85. This surpasses the accuracy of MAS approach when applied to a larger dataset. This improvement is particularly significant for malware families that previously exhibited high false negative rates in FR-Study. Altogether, the main contributions of this chapter are:

- **DroidXPflow:** a novel dynamic analysis approach for Android malware detection that relies on network traffic data collected using DroidXP and ML algorithms.
- **An empirical study:** that brings evidence that DroidXPflow outpeforms the MAS approach. In particular, DroidXPflow can correctly identify malware from families (such as *gappusin*) as the original version of the MAS approach approach can not.

[2]Hereafter, when we use the term app pair(s), we refer to original and repackaged versions of an Android application

As an implication of this research, we argue that robust malware detection sandboxes should monitor not only calls to sensitive APIs but also network traffic to detect potential malicious behavior in Android apps.

6.2 DroidXPflow

In this section, we detail some design decisions related to DroidXPflow, which uses DroidXP [24] to collect network traffic of Android apps while using a test case generation tool and leverages ML algorithms to classify the apps as malware or non-malware. Since DroidXPflow relies on ML algorithms, we organize this section according to typical ML stages: traffic collection, feature extraction, and model training/classifier (see Figure 6.1).

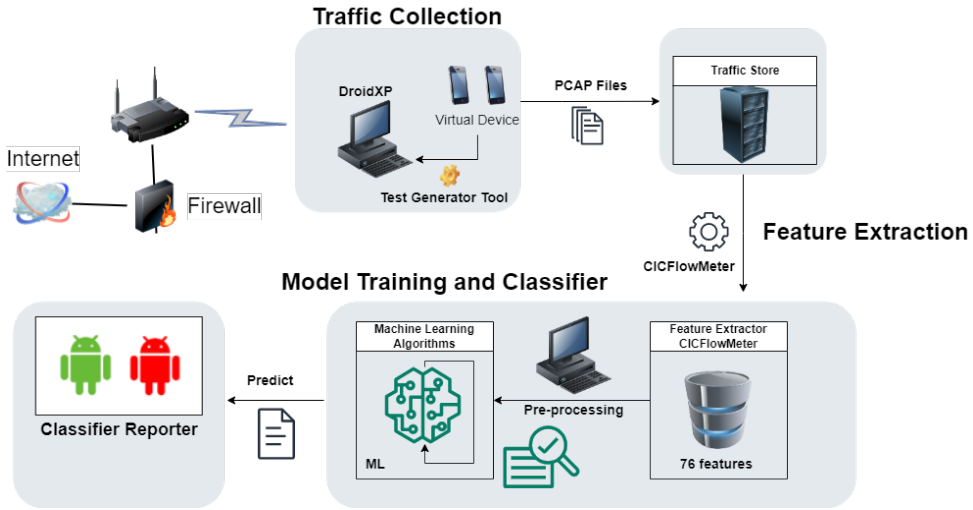


Figure 6.1: Architecture of the DroidXPflow designed for malware detection

6.2.1 Traffic Collection

The DroidXP [24] tool was initially designed to compare test case generation tools in terms of identifying malicious app behaviors using the MAS approach. This makes it a relevant tool for automating data collection using dynamic analysis. For our study, we extended the original version of DroidXP by adding new features to its execution phase. This extension (DroidXPflow) now uses the TcpDump tool to collect both inbound and outbound network traffic. The DroidXPflow extension allows us to capture data for network traffic in the PCAP format [147]. A PCAP file contains copies of network packets, enabling dynamic analysis of both payloads and packet headers [147]. Since storing and processing PCAP

files is a resource-intensive process, DroidXPflow only performs an analysis of network flow segments, rather than analyzing all data available in the PCAP files. As such, we pre-process the PCAP files to extract the most relevant features for our study, using the CICflowMeter tool [148].

CICFlowMeter extracts feature sets in CSV format from the corresponding PCAP files, combining them into a single file that contains a total of 86 features, including flow duration, destination port, number of transmitted bytes, and so on. DroidXPflow considers all features except for eight of them: Flow ID, Source IP, Destination IP, Source Port, Source MAC, Destination MAC, Protocol, and Timestamp. This results in a total of 78 features being analyzed.

During traffic collection, we discard observations where the "Destination Port" does not correspond to the traffic from the HTTP (and its variants) and DNS protocols. The destination port numbers help categorize the protocols utilized in network traffic. The ports corresponding to these services are usually open in firewalls and other security mechanisms to make services accessible to clients. Hence, malware also uses them to get to target services and abuse them. Furthermore, previous studies have demonstrated the relevance of these destination ports in classifying malicious behavior [149, 150], as they are often exploited for purposes such as communication with Command & Control (C&C) servers, carrying out exploits, or blending in with normal traffic [150]. Commonly targeted ports include 80 (HTTP), 443 (HTTPS), and 53 (DNS). In our study, these ports account for 71.80% of the total network traffic captured in our experiment (see Table 6.1). Considering the significance of the Destination Port feature, we opted to filter out network traffic data with just these three destination ports identified in our analysis.

Table 6.1: The three most relevant destination ports in our study.

Port	Description	Occurrences
443	Hypertext Transfer Protocol Secure	1,275,293
53	Domain Name System	641,965
80	Hypertext Transfer Protocol	38,830

6.2.2 Feature Extraction

Selecting relevant features (i.e., feature extraction) is crucial for achieving strong predictive performance in machine learning (ML) models [151, 152].

In this regard, DroidXPflow computes seven additional statistical features (count, minimum, maximum, average, median, variance, and skewness) for each of the 76 original numeric features generated by CICFlowMeter from the PCAP files. These additional statistical features are then fed into the model. At the end of this stage, our ML models

consider a total of 1,598 features (76 numeric features from CICFlowMeter × 3 protocols (destination ports) × 7 statistics), plus the hash ID of the Android apps and the label (i.e., malware or non-malware).

6.2.3 Model Training and Classifier

To compare the performance of the ML algorithms, we train the models based on all 1,598 features extracted from our samples, and later used for malware classification. In our learning-based classification procedure, we split the dataset into a training set consisting of 70% of the samples and a testing set consisting of 30%, randomly selected from the initial dataset. The same set of samples, selected for both training and testing, were used for all the ML algorithms explored. The testing set was used to evaluate the performance of the models, in terms of Recall, F1-score, and Area Under the Curve (AUC) metrics. In our study, we selected six classic ML algorithms commonly used for malware detection (binary classification). There by, we compared the performance of the following ML algorithms:

- Linear Discriminant Analysis (LDA),
- Quadratic Discriminant Analysis (QDA),
- Logistic Regression (LR),
- Random Forest (RF),
- Multi-layer Perceptron (MLP) and
- Support Vector Machines (SVM)

6.3 Empirical Assessment

In this section, we present an empirical assessment of DroidXPflow, which we use to classify repackaged apps as malware or non-malware, using network flow data and ML algorithms. We first characterize the study using the *Goal, Questions, and Metrics* approach (Section 6.3.1); and then present the dataset used in our study (Section 6.3.2).

6.3.1 Goal, Questions, and Metrics

The **goal** of this empirical study is to understand how effective is DroidXPflow in malware detection. To achieve this goal, we investigate the following research **questions**:

(RQ1) What is the performance of ML algorithms for malware detecting using the DroidXPflow approach, in terms of F1-score?

(RQ2) How much gain we obtain on the malware detecting accuracy when we combine the MAS approach and DroidXPflow?

(RQ3) How effective is DroidXPflow at detecting malware from specific families where the MAS approach demonstrated poor performance?

(RQ4) How effective is DroidXPflow at correctly detecting malware from samples where `avclass2` tool cannot identify specific families?

To address these research questions, we first conduct an exploratory study to answer (RQ1) by comparing the performance of the machine learning (ML) algorithms described in Section 6.2.3. We utilize all $1,640$ features from our dataset and employ optimized hyper-parameters for each algorithm. The hyper-parameters were optimized by systematically varying them through cross-validation [153] on the training data. Cross-validation evaluates the model's performance for different combinations of hyper-parameters, ensuring that the model's performance is generalizable and not dependent on a specific train-test split [154]. After applying cross-validation, the optimal set of parameters recommended for each algorithm is presented in Table 6.2. This procedure ensures fair and consistent testing across all algorithms, providing a more reliable basis for answering (RQ1). Finally, to address the remaining research questions-RQ2, RQ3, and RQ4—we utilize the best-performing ML algorithm identified during the exploration of RQ1.

We also address these questions using standard **metrics** to estimate model performance. We use the same procedures we follow in our previous study in Chapter 5. That is, we label the repackaged versions of the apps in our dataset based on the outcomes from `VirusTotal`—a widely used platform that relies on a collection of malware engines to track malicious programs. Using `VirusTotal`, we compute true positives, false positives, and false negatives as follows:

- **True Positive (TP)**. DroidXPflow classifies a repackaged version as malware and, according to `VirusTotal`, at least two security engines label the asset as malware. This decision aligns with existing recommendations [155, 41]
- **False Positive (FP)**. DroidXPflow classifies a repackaged version as malware and, according to `VirusTotal`, at most one security engine labels the asset as malware.
- **False Negative (FN)**. DroidXPflow do not classifies a repackaged version as a malware, and according to `VirusTotal`, at least two security engines label the asset as a malware.

6.3.2 Dataset

Our empirical assessment uses the same dataset (`LargeDS`) described in Chapter 5. This dataset contains 5,844 real-world apps from two repositories of repackaged Android apps: (RePack [10] and AndroMalPack [132]). Of these, 1,777 are original versions, and 4,067 are repackaged versions. Multiple repackaged versions of the same original app might appear within the `LargeDS` dataset. According to `VirusTotal`, at least two security engines classified 2,886 out of the 4,067 repackaged apps as malware. Accordingly, we labeled 2,886 of the repackaged apps as malware (70.96%) and 1,181 as non-malware (29.04%). The `LargeDS` dataset contains several features related to the apps, including information about malware families and a similarity score between the original and repackaged versions of each app. Further details about the `LargeDS` dataset and how they were obtained can be found in Chapter 5.

Since we aim to classify repackaged versions of apps as malware and non-malware, we only collect the network traffic data generated exclusively by the repackaged samples from the `LargeDS` dataset. To capture the network traffic, we execute the repackaged apps in our dataset using the same DroidXP configuration employed in the BLL-Study. Specifically, we use DroidBot [20] as the test case generation tool and collect the network traffic for three minutes.

The outcomes of these executions resulted in multiple PCAP files (4,067), one for each repackaged app from `LargeDS`, as mentioned in Section 6.2.1. A PCAP file contains copies of network packets, enabling the analysis of both payloads and packet headers [147]. Following this, we applied the feature extraction procedure described in Section 6.2.2 to build the flow dataset (`FlowDS`) used in our experiment. The dataset contains a total of 1,640 features, which are derived from a combination of 78 features extracted by CICFlowMeter, three protocol-related features (destination ports), and seven statistical features. The dataset also includes the hash ID of the Android apps and the corresponding malware label (i.e., malware or non-malware).

6.4 Results

In this section, we present the results of our research. First, in Section 6.4.1, we present a comparison of the performance of different ML algorithms in classifying the repackaged versions of apps in `FlowDS` as either malware or non-malware using DroidXPflow. In Section 6.4.2, we present the results of combining the MAS approach with DroidXPflow, while, in Section 6.4.3, we discuss how effectively our approach classifies malware families for which the MAS approach exhibits poor performance.

Table 6.2: Parameters suggested for each algorithm by cross-validation technique
.

Algorithm	Parameter	Grid Space	Value
LDA	Tolerance for singular value decomposition	[1e-5:10]	0.0001
	Solver method	[svd, lsqr, eigen]	eigen
	Shrinkage	[None, auto, 0:1]	0.5540245860
QDA	Store Covariance	[True, False]	True
	Regularized Covariance	[0:1]	0.9997110797
LR	Inverse of regularization strength	[1e-7:10]	0.0001071493
	Solver algorithm	[newton-cg, lbfgs, liblinear, sag, saga]	liblinear
	Maximum number of iterations for optimization	[50:500]	100
	Regularization type	[l1, l2, elasticnet, none]	l1
RF	Number of trees in the forest	[10:500]	210
	The min. num. of samples requir. to split an internal node	[2:20]	10
	The min. num. of samples requir. to be at a leaf node	[1:10]	3
	The num. of features to consider for the best split	[auto, sqrt, log2, 0.5, 0.8, 1.0]	$\log_2(features)$
	The maximum depth of the tree	[None, 5, 10, 20, 30, 50, 100]	5
MLP	Activation function	[relu, tanh, logistic]	relu
	Optization algoritm	[adam, lbfgs, sgd]	adam
	The maximum number of iterations	[200, 500, 1000]	1000
SVM	Regularization parameter	[0.01:100]	5.0
	Kernel type	[linear, poly, rbf, sigmoid]	rbf
	Kernel coefficient	[scale, auto,0.001:1.0))	scale
	Degree of the polynomial kernel	[2:6]	3

6.4.1 Comparison of Machine Learning Algorithms

As discussed in Section 6.2, DroidXPflow extends DroidXP to collect network flow information from apps during test case generation campaigns. To answer our first research question, we conduct an exploratory data analysis that compares the performance of ML algorithms using our `FlowDS` dataset.

For this first study, we execute the algorithms using their optimal hyper-parameter configurations, which we obtained using cross-validation. Cross-validation is an approach that divides the dataset into k equal parts, known as folds. For each possible combination of hyper-parameters (e.g., for a Random Forest algorithm, this could include the number of trees, maximum depth of the tree, etc.), the following steps are repeated:

(s1) Train the model using the (k-1) folds

(s2) Use the remaining fold to evaluate the performance in terms of F1-score.

(s3) Repeat this process k times, each time using different (k-1) folds to train and the remaining fold to validate.

(s4) After training and validating the model k times, calculate the average performance among all folds for a specific combination of hyper-parameters.

(s5) Repeat steps s1 to s4 for all possible combinations of hyper-parameters.

Finally, the best combination of hyper-parameters that yields the highest average performance is selected. The final values for each parameter, relative to all explored algorithms, are presented in Table 6.2.

After identifying the best hyper-parameters, we train the algorithms using the set of 2,847 samples (70%) from the `FlowDS` dataset and test it on 1,220 samples (30%)—using the identified optimal hyper-parameters. Ultimately, the Random Forest algorithm outperformed the others when considering metrics Recall, F1-score, and Area Under the Curve (AUC). Table 6.3 presents the results.

Table 6.3: Performance of the ML algorithms to classify the app as malware or non-malware using network flow data from the `FlowDS`.

Algorithm	Precision	Recall	F1-score	AUC
LDA	0.73	0.97	0.83	0.72
QDA	0.74	0.96	0.83	0.86
LR	0.75	0.92	0.83	0.81
RF	0.75	0.98	0.85	0.88
MLP	0.78	0.85	0.82	0.82
SVM	0.76	0.96	0.84	0.84

Since the RF algorithm outperformed the others, in the following sections, when we present the performance of DroidXPflow, we will describe its usage with the Random Forest algorithm.

Finding 6 *The RF algorithm outperforms the other algorithms, achieving higher values among the F1-score and Area Under the Curve metrics when exploring all features and using the algorithms with optimal hyper-parameters configurations.*

6.4.2 A comparison between DroidXPflow and the MAS approach

Here, we compare the performance of DroidXPflow and the MAS approach on malware classification. This comparison also relies on standard metrics (recall, precision, and F1-score), and we present the results using the same 30% of samples (1,220) from the `FlowDS` dataset that we used for the tests in Section 6.4.1. Our results show that the vanilla MAS approach for malware identification achieves significantly lower F1-score compared to the DroidXPflow framework. That is, the MAS approach achieves an F1-score of 59%, while the DroidXPflow achieves an F1-score of 85% when using the RF algorithm.

In more detail, considering the 1,220 apps in the testing sample (30% of the `FlowDS`), the MAS approach classified a total of 452 repackaged apps as malware (37.04%) of

Table 6.4: Performance of both strategy on `FlowDS` (1220 samples).

Approach	TP	FP	FN	Precision	Recall	F_1
MAS approach	388	64	488	0.86	0.45	0.59
DroidXPflow	854	278	19	0.75	0.98	0.85
DroidXPflow and MAS approach	872	289	4	0.75	0.99	0.86

the total number of repackaged apps in the testing sample), where the repackaged app version calls at least one additional sensitive API. Note that the MAS approach fails to correctly classify 488 assets as malware (FN) (first row of Table 5.3) and wrongly labels the repackaged version of 64 apps as malware (FP). This leads to poor performance in terms of F1-score, indicating that, when considering the `FlowDS`, the F1-score of the MAS approach using DroidBot as the test generation tool is 59%.

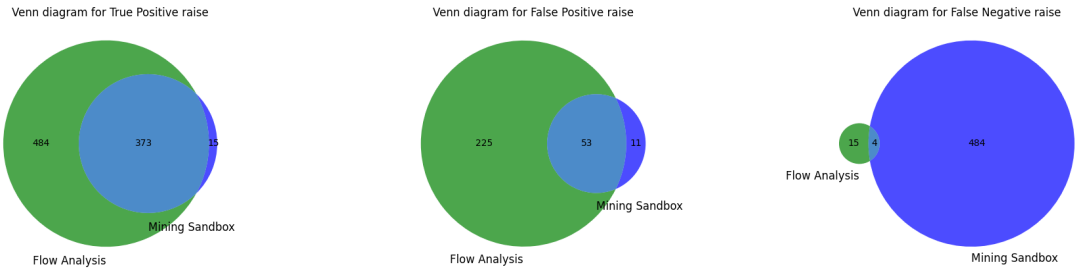
In contrast, DroidXPflow classified 1,132 apps as malware but failed to label 19 assets as malware (FN) correctly. In addition, DroidXPflow wrongly labeled (FP) the repackaged versions of 278 samples as malware (second row of Table 5.3). DroidXPflow led to a better performance than the MAS approach, with an F1-score of 85%. Based on these results, we can conclude that DroidXPflow outperforms the MAS approach when exploring `FlowDS`.

Finding 7 *The experimental results show that DroidXPflow outperforms the MAS approach, with F1-score of 0.85, compared to 0.59 for the MAS approach.*

We further investigate the benefits of combining both approaches (MAS approach and DroidXPflow). In this case, a true positive (TP) happens whenever at least one of the approaches correctly identifies a malicious sample. In contrast, a false negative (FN) occurs when both approaches incorrectly classify a malicious sample as benign, while a false positive (FP) happens when at least one of the approaches incorrectly classifies a benign sample as malicious. This strategy correctly classified 872 repackaged apps as malware (TP) and reduced the number of false negatives (FN) to 4. However, it also increased the number of false positives (FP) to 289. In summary, the results show that combining both techniques slightly increases the recall and F1-score metrics (third row of Table 5.3).

Finding 8 *Combining the MAS approach with DroidXPflow leads to a marginal benefit in terms of recall (from 0.98 to 0.99) and F1-score (from 0.85 to 0.86).*

To understand the benefits of each method, we also analyze the contribution of both methods in detecting malicious samples. We show Venn diagrams highlighting the sets of True Positives (TP), False Positives (FP), and False Negatives (FN) for each technique in Figure 6.2, which reveals that different approaches contribute differently to the final detection result. For instance, in the first Venn diagram of Figure 6.2, we present the



(a) True Positive raise (b) False Positive raise (c) False Negative raise

Figure 6.2: Contribution to the final detection result

set of True Positives (TP) for our samples. As one can see, 484 samples were identified solely by DroidXPflow—most of them from the *gappusin* family (316 samples)—while both approaches detected 373 malware, mainly from *gappusin*, *airpush*, and *dowgin*. The MAS approach correctly detected 15 malware not detected by DroidXPflow, most without family classification (four samples) and from the *revmob* family (three samples).

The second Venn diagram presents the sets of False Positives (FP). The diagram shows that the MAS approach has a lower contribution to FPs compared to DroidXPflow, which can explain its precision in Table 5.3. For 225 benign samples, the DroidXPflow approach wrongly classified a normal network flow as malicious. However, its precision remains high due to the increase in True Positives (TP). Finally, the third Venn diagram shows the samples that both approaches failed to classify as malicious. In this case, the MAS approach leads to a high amount of False Negatives (FN) (484 samples), most of them from the *gappusin* family (316 samples), which can mainly be detected by DroidXPflow due to their malicious network behavior. Among the four samples that were not detected by either approach, they do not have a family classification. According to `VirusTotal`, among the 60 antivirus engines available, at most four can detect them, characterizing these samples as complex and difficult to classify as malware.

6.4.3 Detection Performance based on Malware Family

Categorizing malware into distinct types is a common practice. For example, Android malware can be divided into adware, trojan, spyware, etc. These categories can be further subdivided into specific malware families based on unique characteristics, attack methods, and objectives [54].

Given that certain malware families, particularly those relying on command-and-control communication, demonstrate substantial network activity, evaluating their detection performance through network traffic analysis can offers valuable insights. This approach complements existing detection methods by incorporating network behavioral

signatures, which were not systematically explored in the FR-Study. In this section, we present the results of our experiment, considering first the samples for which we could identify the malware family (using the `avclass2` tool [136]); and then considering the samples for which we were not able to determine the family.

In the FR-Study, the authors showed that the MAS approach has a high false negative rate for the *gappusin* and *revmob* families. Specifically, it failed to correctly identify 87.50% of the samples from the *gappusin* family and 44.44% of the samples from the *revmob* family as malware. Among the 67 samples evaluated from the *revmob* family, DroidXPflow identified 64 apps (95.52%) as malware, while for samples from the *gappusin* family, it identified 371 samples (99.73%) as malware. Therefore, DroidXPflow effectively detects samples with malicious network behaviors for which the MAS approach failed to identify as malware correctly.

Considering the *gappusin* family, the reverse engineering discussed in the FR-Study reveals that their samples automatically communicate with remote servers to download and install other apps or adware without the user's knowledge [156]. This network traffic might explain the superior performance of DroidXPflow to classify samples of the *gappusin* family as malware correctly.

Finding 9 *DroidXPflow proved to be efficient in identifying samples from malware families where the MAS approach demonstrated poor performance, as shown when considering samples from* gappusin *and* revmob *families, popular malware families.*

Additionally, it is worth noting that among all families examined, DroidXPflow correctly identified 100% of samples as malware in 65 of these families. This result shows the effectiveness of our approach for other malware families investigated as well. For instance, Table 6.5 presents the DroidXPflow detection performance for the families with at least four samples in our dataset, as well as the classification of the samples into five malware categories: ad fraud, adware, spyware, trojans, and SMS malware.

Table 6.5 also shows that most samples belong to the adware and ad fraud categories, which aggressively display ads, generate fraudulent ad clicks, and track user behavior [157]. The other categories are equally harmful, though: SMS malware may cause financial losses by sending unauthorized messages, spyware collects sensitive user data for malicious purposes, and trojans provide remote access and control over infected devices.

Among the samples from our `FlowDS`, at least two security engines identified 80 as malware, even though the `avclass2` tool could not determine their families, perhaps because they were recently discovered. Since new malware emerges daily, accurately classifying recent malicious apps into their respective families is both challenging and time-consuming [158, 159].

Table 6.5: Detection Rate of the Families with at least four samples.

Category	Family	Samples	Detected	%
Ad Fraud	gappusin	372	371	99,73
	dowgin	69	67	97,10
	kyvu	4	4	100,00
Adware	revmob	67	64	95,52
	airpush	44	43	97,72
	youmi	28	28	100,00
	kuguo	22	22	100,00
	leadbolt	15	15	100,00
	adwo	10	10	100,00
	apptrack	10	10	100,00
	domob	9	9	100,00
	appsgeyser	6	6	100,00
	admogo	6	6	100,00
	pircob	5	5	100,00
	cimsci	4	4	100,00
	dnotua	3	3	100,00
Spyware	igexin	7	7	100,00
	cnzz	4	4	100,00
Trojan	torjok	8	8	100,00
SMSmalware	smsreg	29	28	96,55
None	unknown	80	72	90,00

Although specific families were unknown at the time of this research, DroidXPflow flagged 72 of these samples (90.00%) as malware. Based on these results, we conclude that while DroidXPflow can identify these samples as malware, it has the lowest recall rate for samples without family classification.

Finding 10 *Even for malicious apps without family classification, DroidXPflow can correctly identify them as malware based on their suspicious network activities. However, the false negative rate (FN) is higher for these samples compared to those with family identification.*

6.5 Discussion

The previous section demonstrated the efficacy of the DroidXPflow for detecting malware in network traffic. In this section, we address the research questions posed in Section 6.3, presenting the implications of our results, and discussing certain limitations that cannot be ignored. These limitations also highlight areas for future research.

6.5.1 Research Questions and Analysis

The assessment of our method in the previous section allows us to answer the research questions as follows:

1. **Machine Learning Algorithms Analysis (RQ1).** Our experimental findings provide evidence that, among all the ML algorithms investigated, the Random Forest (RF) algorithm outperformed the other five algorithms explored, using the best hyper-parameters suggested by the cross-validation technique.

2. **Performance Gain in Android Malware Classification when we combine the MAS approach and DroidXPflow (RQ2).** Our study indicates that the performance of DroidXPflow is superior when compared to MAS approach in malware identification. However, combining both approaches leads to only a marginal gain in performance (in terms of F1-score).

3. **Detection Performance on family where MAS approach demonstrated low performance (RQ3).** We confirm that DroidXPflow presents a better performance for samples from families where the MAS approach has a poor performance. For example, samples from the *revmob* and *gappusin* families achieved correct identification rates above 95% using DroidXPflow. These malware families are primarily characterized by downloading adware without the user's knowledge, automatically connecting to and interacting with remote servers [156].

4. **Detection Performance on samples without family identification (RQ4).** In malware samples where the `avclass2` tool could not classify the family, DroidXPflow achieved an acceptable accuracy (F1-score) of over 90%. However, we observed that this rate is lower than the performance for identifying samples in which the family was known. In other words, without identifying the unique traits or attack methods that characterize a specific family, DroidXPflow can still detect malicious network activity that deviates from normal behavior.

6.5.2 Implications

Previous studies [3, 21, 26] incorrectly identified the MAS approach as a solution with reasonable performance, based on results from a limited dataset composed of fewer than 20 malware families. In contrast, FR-Study reported negative results for the MAS approach when using a more representative dataset, which included a greater variety of malware

families. These families were responsible for a higher false negative rate, ultimately compromising the performance of the MAS approach.

Our work on DroidXPflow addresses this problem, presenting an approach based on network flow analysis with ML support. Our solution proved to be efficient in detecting different malicious behaviors and reducing the number of false negatives. More importantly, DroidXPflow can identify more malware that use polymorphism or obfuscation to evade detection [160], but exhibit high and suspicious interactions with the network.

Our study also reveals that for 15 samples, DroidXPflow fails to detect their malicious behavior, while the MAS approach successfully identified them as malware. This result suggests that Network Traffic Analysis is not a complete solution, highlighting the importance of combining both approaches. Among all families explored, eight had samples that DroidXPflow do not identify as malicious; however, they were identified as malicious by MAS approach. Examples include the *Dowgin* family (2 samples out of 69) and the *Revmob* family (3 samples out of 67), where only the MAS approach was able to identify them correctly as malware. Furthermore, our previous results show that MAS approach can correctly label as malware 100% samples from the *Airpush* family. In this case, DroidXPflow also classifies all malware samples and confirms their maliciousness. This demonstrates that the current state-of-the-art Mining Sandbox techniques remain effective for certain malware families.

6.5.3 Limitations

The previous results of our empirical assessment show that DroidXPflow is a practical approach for malware detection. However, as with any empirical study, some potential threats are worth highlighting.

Training set. The `FlowDS` contains 2,886 malware samples, comprising 116 families. However, since new malware appears daily, we believe there are still malware families that cannot be detected by DroidXPflow. This is a typical limitation in the malware detection literature, although our dataset is competitive with previous studies (see Table 2.1). To address this issue in future studies, we propose expanding the dataset continuously, including experimenting with additional detection models. The malware detection capability improves as the size of the training samples increases, enabling the solution to detect more types of malware.

Malicious behaviors triggered. During the execution phase of DroidXP, it restarts the explored apps to activate the malicious behavior of a malware. However, it is possible that not all malicious activities were fully triggered by our testing tool, as some behaviors may require real user interaction to be activated. Bao et al. [3] provide evidence that DroidBot outperforms other test generation tools by uncovering many potential malicious

behaviors. However, since this research, different tools for test case generation have emerged [161, 162], which tend to perform more effectively and could make the collected traffic resemble real-world scenarios more closely. Furthermore, since we used an Android emulation environment, it is possible that some malware could detect this situation and avoid triggering their malicious behaviors, thus affecting the network traffic collection process. Besides that, integrating DroidXPflow with Random Forest led to a recall of 98%.

In the future, we plan to explore more recent test generation tools, as presented in Chapter 2, that could cover a broader range of app behaviors. Additionally, we intend to incorporate real devices into the traffic collection to detect better malware that can bypass emulators.

6.6 Conclusions

In this chapter, we introduced DroidXPflow, a framework for detecting Android malware using Network Traffic Analysis with the support of Machine Learning algorithms. As a first step, we created a dataset of network traffic (`FlowDS`) from the execution of 4,067 repackaged apps, extracted from the dataset presented in FR-Study. We then used DroidXPflow to investigate whether our solution could overcome the limitations of the MAS approach, especially relating to the detection of malware families that heavily interact with networks. Our evaluation demonstrates that DroidXPflow achieves a good performance in detecting Android malware, with an F1-score of 0.85. Although DroidXPflow builds upon the results of the state-of-the-art Mining Sandbox, we show that it is not a complete solution. Our evaluation reveals that there are samples in `FlowDS` that DroidXPflow fails to flag as malicious, whereas the MAS approach succeeds. We also highlight the limitations posed by our malicious sample quantity and discuss the importance of the number of samples used for training, as this can affect the accuracy of ML algorithms and, consequently, the effectiveness of our approach. In future work, we plan to collect, train, and analyze more malware samples to improve our malware detection solution by developing more sophisticated models. Additionally, we intend to explore more recent test generation tools that can better simulate user input, thereby making the collected traffic more closely resemble real-world scenarios.

Chapter 7

Conclusions

As presented in Section 2.3.2, the MAS approach is a popular technique used to isolate Android apps in order to analyze their behavior and identify vulnerabilities. It has been extensively studied in prior research, including the BLL-Study. The study reported that more than 70% of the malware investigated in their dataset could be detected by sandboxes built through the synchronous execution of test case generation tools (i.e., dynamic analysis). However, our first study presented in Chapter 4 revealed that this performance is only achieved when we enable the static analysis components from DroidFax. This finding demonstrates that the effectiveness of the MAS approach stems from the combined use of both analyses (static and dynamic), with the latter being facilitated by test generation tools. We review the literature and present, for the first time, empirical evidence that the MAS approach benefits from the combination of both dynamic and static analysis.

In Chapter 5, we investigated the strengths and limitations of the MAS approach for detecting repackaged malware using a more diverse dataset compared to the one used in the BLL-Study. Our results reveal that the performance of the MAS approach drops significantly when applied to our comprehensive dataset, highlighting the limitations of the MAS approach, particularly when scaled. These findings pave the way for our third study, presented in Chapter 6, in which we propose a method for detecting Android malware by Network Traffic Analysis with ML support. In this study, we use the same dataset from Study 2 to investigate whether our proposed method can overcome the limitations of the MAS approach pointed out in Chapter 5. The study demonstrates that the Network Traffic Analysis with the support of the Random Forest algorithm achieves strong performance in detecting mobile malware, with an F1-score of 0.85. Although this is an interesting achievement, we show that Network Traffic Analysis is not a complete solution, as there are samples in the dataset that the proposal fails to classify as malware, but which the MAS approach successfully identifies.

As briefly outlined above, we have sought to contribute to the ongoing "cat-and-mouse" game, hoping that our thesis empirically expands the current understanding of MAS approach for malware detection. We examined its primary strengths and limitations and proposed solutions to address the identified weaknesses. In the following sections, we present the contributions of this thesis and lay the groundwork for future research on Android malware and its detection.

7.1 Contributions and Findings

In this thesis, we investigate how the MAS approach can be utilized to detect malware and prevent privacy leaks in Android apps, such as the unauthorized disclosure of personal data, network MAC addresses, IMEI numbers, and similar sensitive data. We examine the limitations of the MAS approach and propose alternative solutions to overcome these challenges. As an initial step toward achieving these objectives, we develop and release several artifacts to support academic research, which we briefly outline below.

DroidXP: [24] An extensible tool that allows researchers and practitioners to easily add and evaluate test generation tools. With support from DroidFax, it instruments each APK file and collects data during app execution while a test tool (e.g., Droidmate or Droidbot) is running. DroidXP is currently available in an open source repository [1].

DroidXPflow: It extends the original DroidXP framework by integrating additional dynamic analysis techniques, in particular network traffic capture during app execution, to enhance malware detection. While DroidXP primarily focused on sensitive API calls, DroidXPflow enriches the DroidXP outcomes with behavioral data, which is then analyzed using machine learning (ML) models trained on a pre-existing knowledge base. This extension enhances detection accuracy, increases resistance to evasion techniques (e.g., code obfuscation), and reduces both false positives (FPs) and false negatives (FNs). The DroidXPflow scripts are publicly available in an open-source repository [2] for the academic community to support future research.

These artifacts proved essential to our thesis, enabling us to both validate earlier work and perform new explanatory studies on the MAS approach, as described below:

[1] github.com/droidxp/benchmark
[2] github.com/droidxp/ML

7.1.1 Study 1: What is the impact of static analysis on the MAS approach?

- This thesis presents the effectiveness of MAS approach in classifying malicious apps with the combination of dynamic analysis, enabled by test generation tools, and static analysis, achieved through instrumentation using the Droidfax algorithm.
- Our work also demonstrates that the performance of FlowDroid static analysis algorithms in identifying malicious behavior is comparable to the performance of MAS approach supported by dynamic analysis, i.e., without the support of DroidFax static analysis algorithms.
- Incorporating static analysis tools like FlowDroid and DroidFax into the MAS approach enhances its effectiveness in classifying Android apps as malware correctly.

7.1.2 Study 2: Does the accuracy of the MAS approach scale at a large and more diverse dataset than previous studies?

- We demonstrate that only a few sensitive APIs are responsible for the majority of malicious code inserted in malware apps, with the most commonly inserted API providing access to telephony services.
- We extend the state-of-the-art knowledge on the effectiveness of the MAS approach, revealing that, on a more diverse dataset, the performance of the MAS approach is substantially lower compared to what was previously observed.
- We present evidence of a correlation between the Similarity Score, which measures the likeness between the original and repackaged versions of an app, and the performance of the MAS approach. Our findings suggest that a higher Similarity Score reduces the likelihood of successful malware classification by the MAS approach
- We bring evidence on the high influence of malware families on the accuracy of the MAS approach. That is, we report that the MAS approach fails to correctly identify most samples from the *gappusin* and *revmob* families as malware, revealing a key reason for the low recall of the MAS approach in the diverse dataset. This finding highlights the need for additional techniques to support the MAS approach in malware identification.

7.1.3 Study 3: Does DroidXPflow outperform the original MAS approach, improving its malware detection capabilities?

- The study shows that Network Traffic Analysis, supported by ML algorithms, outperforms the MAS approach, proving to be an effective strategy to enhance malware classification.
- The study also provides evidence that the Random Forest algorithm is the best option for analyzing network flow with ML, achieving a higher F1-score (0.85). Furthermore, our assessments also highlight that DroidXPflow significantly enhances malware detection performance for the families identified in Study 2, which presented the lowest recall rates but demonstrated high levels of malicious network activity. DroidXPflow effectively leverages and improves these recall rates.
- Our thesis demonstrates that combining both techniques (i.e., DroidXPflow and the MAS approach) yields a slight improvement in recall (increasing from 0.98 to 0.99) and, consequently, a marginal increase in the F1-score (from 0.85 to 0.86).
- Last but not least, study 3 provides evidence that DroidXPflow is most effective in classifying malware among samples with a known family. However, it still performs well on samples without known family by identifying suspicious network activities that deviate from patterns typically classified as normal.

7.2 Future Work

In Chapter 2, we introduce several test generation tools capable of generating test cases for the MAS approach, and in Chapter 4, we show that Droidbot performs the best in this regard. However, for future work, it would be valuable to explore more recent test generation tools that could cover a broader range of app behaviors and potentially surpass Droidbot's performance. Additionally, incorporating real devices into the data collection process could prove beneficial, as this approach might help identify malware samples that are capable of detecting and bypassing emulated environments.

Furthermore, as discussed in Section 5.4.2, the versatility of the Java Native Interface (JNI) has introduced new challenges. Malware authors are increasingly leveraging the native layer to conceal malicious code. Our research indicates that the current state-of-the-art in the MAS approach, often overlooks native calls to sensitive resources. As future work, we believe it is important to conduct research on the use of native components to identify multiple suspicious activities associated with JNI code. Section 5.4.2 also highlights insecure coding practices for Android development, emphasizing the need for

robust safeguards against common vulnerabilities such as improper JNI usage, insecure native library loading, and download of apps at runtime. Future work should focus on providing developers with actionable guidelines to prevent security vulnerabilities in Android apps, as illustrated in listings 5.3 and 5.5, by extracting common vulnerability patterns. These patterns would help developers avoid security pitfalls while establishing best practices for building more secure Android apps.

Finally, as with all ML strategies, it is important to keep the training set continuously updated. We envision further investigation with a larger number of malware samples that could cover more malware families, as the malware detection capability improves with the increase in the size of the training set. The processes of data collection, training, and analysis must be ongoing to continuously improve malware classification. It is also important to highlight that continuous updates to malware/non-malware classifications and family categorizations are also essential, using standardized tools like `VirusTotal`, since these threat classifications evolve over time.

Reference

[1] Krajci, Iggy and Darren Cummings: *Android on x86: An Introduction to Optimizing for Intel® Architecture*. Springer Nature, 2013. xv, 9, 10

[2] Jung, Jin-Hyuk, Ju Young Kim, Hyeong-chan Lee, and Jeong Hyun Yi: *Repackaging attack on android banking applications and its countermeasures*. Wirel. Pers. Commun., 73(4):1421–1437, 2013. https://doi.org/10.1007/s11277-013-1258-x. xv, 11

[3] Bao, Lingfeng, Tien-Duy B. Le, and David Lo: *Mining sandboxes: Are we there yet?* In Oliveto, Rocco, Massimiliano Di Penta, and David C. Shepherd (editors): *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*, pages 445–455. IEEE Computer Society, 2018. xv, 1, 2, 3, 14, 17, 18, 19, 22, 27, 31, 32, 33, 36, 41, 54, 56, 58, 59, 60, 61, 62, 65, 66, 67, 75, 76, 77, 78, 81, 82, 94, 95

[4] Baldoni, Roberto, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi: *A survey of symbolic execution techniques*. ACM Comput. Surv., 51(3):50:1–50:39, 2018. https://doi.org/10.1145/3182657. xv, 23, 24, 25

[5] Bhat, Parnika and Kamlesh Dutta: *A survey on various threats and current state of security in android platform*. ACM Comput. Surv., 52(1):21:1–21:35, 2019. https://doi.org/10.1145/3301285. 1, 12, 19

[6] *International data corporation (idc)*. https://www.idc.com/promo/smartphone-market-share/os. Accessed: 2021-03-17. 1

[7] Faruki, Parvez, Ammar Bharmal, Vijay Laxmi, Vijay Ganmoor, Manoj Singh Gaur, Mauro Conti, and Muttukrishnan Rajarajan: *Android security: A survey of issues, malware penetration, and defenses*. IEEE Commun. Surv. Tutorials, 17(2):998–1022, 2015. https://doi.org/10.1109/COMST.2014.2386139. 1, 12, 26, 35

[8] Sufatrio, Darell J. J. Tan, Tong-Wei Chua, and Vrizlynn L. L. Thing: *Securing android: A survey, taxonomy, and challenges*. ACM Comput. Surv., 47(4):58:1–58:45, 2015. https://doi.org/10.1145/2733306. 1, 26, 35

[9] Le, Tien Duy B, Lingfeng Bao, David Lo, Debin Gao, and Li Li: *Towards mining comprehensive android sandboxes*. In *2018 23rd International conference on engineering of complex computer systems (ICECCS)*, pages 51–60. IEEE, 2018. 1, 3, 81

[10] Li, Li, Tegawendé F. Bissyandé, and Jacques Klein: *Rebooting research on detecting repackaged android apps: Literature review and benchmark*. IEEE Trans. Software Eng., 47(4):676–693, 2021. https://doi.org/10.1109/TSE.2019.2901679. 1, 13, 14, 58, 61, 87

[11] Arzt, Steven, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick D. McDaniel: *Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps*. In O'Boyle, Michael F. P. and Keshav Pingali (editors): *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pages 259–269. ACM, 2014. https://doi.org/10.1145/2594291.2594299. 1, 23, 35, 39

[12] Costa, Francisco Handrick da, Ismael Medeiros, Thales Menezes, João Victor da Silva, Ingrid Lorraine da Silva, Rodrigo Bonifácio, Krishna Narasimhan, and Márcio Ribeiro: *Exploring the use of static and dynamic analysis to improve the performance of the mining sandbox approach for android malware identification*. CoRR, abs/2109.06613, 2021. https://arxiv.org/abs/2109.06613. 1

[13] Jamrozik, Konrad, Philipp von Styp-Rekowsky, and Andreas Zeller: *Mining sandboxes*. In Dillon, Laura K., Willem Visser, and Laurie A. Williams (editors): *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 37–48. ACM, 2016. https://doi.org/10.1145/2884781.2884782. 1, 2, 3, 14, 16, 17, 26, 35, 36, 38, 54, 59, 81

[14] Krüger, Stefan, Johannes Späth, Karim Ali, Eric Bodden, and Mira Mezini: *Crysl: An extensible approach to validating the correct usage of cryptographic apis*. In Millstein, Todd D. (editor): *32nd European Conference on Object-Oriented Programming, ECOOP 2018, July 16-21, 2018, Amsterdam, The Netherlands*, volume 109 of *LIPIcs*, pages 10:1–10:27. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. https://doi.org/10.4230/LIPIcs.ECOOP.2018.10. 1, 35

[15] Rahaman, Sazzadur, Ya Xiao, Sharmin Afrose, Fahad Shaon, Ke Tian, Miles Frantz, Murat Kantarcioglu, and Danfeng (Daphne) Yao: *Cryptoguard: High precision detection of cryptographic vulnerabilities in massive-sized java projects*. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 2455–2472, New York, NY, USA, 2019. Association for Computing Machinery, ISBN 9781450367479. https://doi.org/10.1145/3319535.3345659. 1, 35

[16] Jamrozik, Konrad and Andreas Zeller: *Droidmate: a robust and extensible test generator for android*. In *Proceedings of the International Conference on Mobile Software Engineering and Systems, MOBILESoft '16, Austin, Texas, USA, May 14-22, 2016*, pages 293–294. ACM, 2016. https://doi.org/10.1145/2897073.2897716. 2, 18, 30, 33, 41

[17] *Monkey*. https://developer.android.com/studio/test/monkey. Accessed: 2020-02-10. 2, 18, 34, 41, 59

[18] Amalfitano, Domenico, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M. Memon: *Using GUI ripping for automated testing of android applications.* In Goedicke, Michael, Tim Menzies, and Motoshi Saeki (editors): *IEEE/ACM International Conference on Automated Software Engineering, ASE'12, Essen, Germany, September 3-7, 2012*, pages 258–261. ACM, 2012. https://doi.org/10.1145/2351676.2351717. 2

[19] Hao, Shuai, Bin Liu, Suman Nath, William G. J. Halfond, and Ramesh Govindan: *PUMA: programmable ui-automation for large-scale dynamic analysis of mobile apps.* In Campbell, Andrew T., David Kotz, Landon P. Cox, and Zhuoqing Morley Mao (editors): *The 12th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys'14, Bretton Woods, NH, USA, June 16-19, 2014*, pages 204–217. ACM, 2014. https://doi.org/10.1145/2594368.2594390. 2

[20] Li, Yuanchun, Ziyue Yang, Yao Guo, and Xiangqun Chen: *Droidbot: a lightweight ui-guided test input generator for android.* In Uchitel, Sebastián, Alessandro Orso, and Martin P. Robillard (editors): *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion Volume*, pages 23–26. IEEE Computer Society, 2017. https://doi.org/10.1109/ICSE-C.2017.8. 2, 18, 19, 22, 30, 33, 41, 59, 65, 82, 87

[21] Le, Tien-Duy B., Lingfeng Bao, David Lo, Debin Gao, and Li Li: *Towards mining comprehensive android sandboxes.* In *23rd International Conference on Engineering of Complex Computer Systems, ICECCS 2018, Melbourne, Australia, December 12-14, 2018*, pages 51–60. IEEE Computer Society, 2018. https://doi.org/10.1109/ICECCS2018.2018.00014. 2, 19, 58, 76, 94

[22] Arzt, Steven, 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. https://doi.org/10.1145/2594291.2594299. 4, 37, 44

[23] Cai, Haipeng and Barbara G. Ryder: *Droidfax: A toolkit for systematic characterization of android applications.* In *2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, September 17-22, 2017*, pages 643–647. IEEE Computer Society, 2017. https://doi.org/10.1109/ICSME.2017.35. 4, 5, 19, 28, 36, 61

[24] Costa, Francisco Handrick da, Ismael Medeiros, Pedro Costa, Thales Menezes, Marcos Vinícius, Rodrigo Bonifácio, and Edna Dias Canedo: *Droidxp: A benchmark for supporting the research on mining android sandboxes.* In *20th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2020, Adelaide, Australia, September 28 - October 2, 2020*, pages 143–148. IEEE, 2020. https://doi.org/10.1109/SCAM51674.2020.00021. 5, 19, 36, 38, 63, 82, 83, 98

[25] Choudhary, Shauvik Roy, Alessandra Gorla, and Alessandro Orso: *Automated test input generation for android: Are we there yet? (E).* In Cohen, Myra B., Lars Grunske, and Michael Whalen (editors): *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, pages 429–440. IEEE Computer Society, 2015. https://doi.org/10.1109/ASE.2015.89. 6, 22, 33

[26] Costa, Francisco Handrick da, Ismael Medeiros, Thales Menezes, João Victor da Silva, Ingrid Lorraine da Silva, Rodrigo Bonifácio, Krishna Narasimhan, and Márcio Ribeiro: *Exploring the use of static and dynamic analysis to improve the performance of the mining sandbox approach for android malware identification.* J. Syst. Softw., 183:111092, 2022. https://doi.org/10.1016/j.jss.2021.111092. 6, 19, 59, 75, 76, 78, 94

[27] Merwe, Heila van der, Brink van der Merwe, and Willem Visser: *Verifying android applications using java pathfinder.* ACM SIGSOFT Softw. Eng. Notes, 37(6):1–5, 2012. https://doi.org/10.1145/2382756.2382797. 9

[28] Mirzaei, Nariman, Sam Malek, Corina S. Pasareanu, Naeem Esfahani, and Riyadh Mahmood: *Testing android apps through symbolic execution.* ACM SIGSOFT Softw. Eng. Notes, 37(6):1–5, 2012. https://doi.org/10.1145/2382756.2382798. 9

[29] Felt, Adrienne Porter, Erika Chin, Steve Hanna, Dawn Song, and David A. Wagner: *Android permissions demystified.* In Chen, Yan, George Danezis, and Vitaly Shmatikov (editors): *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*, pages 627–638. ACM, 2011. https://doi.org/10.1145/2046707.2046779. 9, 12, 20, 65

[30] Six, Jeff: *Application Security for the Android Platform.* O'Reilly Media, 2011, ISBN 1449315070, 9781449315078. 10

[31] *Permissions on Android*, May 2023. https://developer.android.com/guide/topics/permissions/overview, visited on 2023-06-09. 11

[32] *Permissions.* https://developer.android.com/guide/topics/permissions. Accessed: 2020-04-25. 11

[33] Wang, Haoyu, Hao Li, Li Li, Yao Guo, and Guoai Xu: *Why are android apps removed from google play?: a large-scale empirical study.* In Zaidman, Andy, Yasutaka Kamei, and Emily Hill (editors): *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*, pages 231–242. ACM, 2018. https://doi.org/10.1145/3196398.3196412. 11

[34] Felt, Adrienne Porter, Kate Greenwood, and David A. Wagner: *The effectiveness of application permissions.* In Fox, Armando (editor): *2nd USENIX Conference on Web Application Development, WebApps'11, Portland, Oregon, USA, June 15-16, 2011.* USENIX Association, 2011. https://www.usenix.org/conference/webapps11/effectiveness-application-permissions. 12

[35] Zhou, Yajin and Xuxian Jiang: *Dissecting android malware: Characterization and evolution.* In *IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA*, pages 95–109. IEEE Computer Society, 2012. https://doi.org/10.1109/SP.2012.16. 12, 13, 81

[36] Felt, Adrienne Porter, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David A. Wagner: *Android permissions: user attention, comprehension, and behavior.* In Cranor, Lorrie Faith (editor): *Symposium On Usable Privacy and Security, SOUPS '12, Washington, DC, USA - July 11 - 13, 2012*, page 3. ACM, 2012. https://doi.org/10.1145/2335356.2335360. 12

[37] Tian, Ke, Danfeng Yao, Barbara G. Ryder, Gang Tan, and Guojun Peng: *Detection of repackaged android malware with code-heterogeneity features.* IEEE Trans. Dependable Secur. Comput., 17(1):64–77, 2020. https://doi.org/10.1109/TDSC.2017.2745575. 13, 81

[38] Hamilton, James Alexander George and Sebastian Danicic: *An evaluation of current java bytecode decompilers.* In *Ninth IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2009, Edmonton, Alberta, Canada, September 20-21, 2009*, pages 129–136. IEEE Computer Society, 2009. https://doi.org/10.1109/SCAM.2009.24. 13

[39] Wu, Dong-Jie, Ching-Hao Mao, Te-En Wei, Hahn-Ming Lee, and Kuo-Ping Wu: *Droidmat: Android malware detection through manifest and API calls tracing.* In *Seventh Asia Joint Conference on Information Security, AsiaJCIS 2012, Kaohsiung, Taiwan, August 9-10, 2012*, pages 62–69. IEEE Computer Society, 2012. https://doi.org/10.1109/AsiaJCIS.2012.18. 13, 16

[40] Wang, Haoyu, Yao Guo, Ziang Ma, and Xiangqun Chen: *Wukong: a scalable and accurate two-phase approach to android app clone detection.* In Young, Michal and Tao Xie (editors): *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*, pages 71–82. ACM, 2015. https://doi.org/10.1145/2771783.2771795. 13

[41] Khanmohammadi, Kobra, Neda Ebrahimi, Abdelwahab Hamou-Lhadj, and Raphaël Khoury: *Empirical study of android repackaged applications.* Empir. Softw. Eng., 24(6):3587–3629, 2019. https://doi.org/10.1007/s10664-019-09760-3. 14, 61, 62, 67, 86

[42] Viennot, Nicolas, Edward Garcia, and Jason Nieh: *A measurement study of google play.* In Sanghavi, Sujay, Sanjay Shakkottai, Marc Lelarge, and Bianca Schroeder (editors): *ACM SIGMETRICS / International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS 2014, Austin, TX, USA, June 16-20, 2014*, pages 221–233. ACM, 2014. https://doi.org/10.1145/2591971.2592003. 14

[43] Kim, Byoung-chir, Kyeonghwan Lim, Seong-je Cho, and Minkyu Park: *Romadroid: A robust and efficient technique for detecting android app clones using a tree structure and components of each app's manifest file.* IEEE Access, 7:72182–72196, 2019. https://doi.org/10.1109/ACCESS.2019.2920314. 14

[44] Neuner, Sebastian, Victor van der Veen, Martina Lindorfer, Markus Huber, Georg Merzdovnik, Martin Mulazzani, and Edgar R. Weippl: *Enter sandbox: Android sandbox comparison*. CoRR, abs/1410.7749, 2014. http://arxiv.org/abs/1410.7749. 14

[45] Seraj, Saeed, Michalis Pavlidis, and Nikolaos Polatidis: *Trojandroid: Android malware detection for trojan discovery using convolutional neural networks*. In Iliadis, Lazaros, Chrisina Jayne, Anastasios Tefas, and Elias Pimenidis (editors): *Engineering Applications of Neural Networks - 23rd International Conference, EAAAI/EANN 2022, Chersonissos, Crete, Greece, June 17-20, 2022, Proceedings*, volume 1600 of *Communications in Computer and Information Science*, pages 203–212. Springer, 2022. https://doi.org/10.1007/978-3-031-08223-8_17. 14

[46] Lekssays, Ahmed, Bouchaib Falah, and Sameer Abufardeh: *A novel approach for android malware detection and classification using convolutional neural networks*. In Sinderen, Marten van, Hans-Georg Fill, and Leszek A. Maciaszek (editors): *Proceedings of the 15th International Conference on Software Technologies, ICSOFT 2020, Lieusaint, Paris, France, July 7-9, 2020*, pages 606–614. ScitePress, 2020. https://doi.org/10.5220/0009822906060614. 14

[47] Wei, Linfeng, Weiqi Luo, Jian Weng, Yanjun Zhong, Xiaoqian Zhang, and Zheng Yan: *Machine learning-based malicious application detection of android*. IEEE Access, 5:25591–25601, 2017. https://doi.org/10.1109/ACCESS.2017.2771470. 14

[48] Zhou, Wu, Yajin Zhou, Xuxian Jiang, and Peng Ning: *Detecting repackaged smartphone applications in third-party android marketplaces*. In Bertino, Elisa and Ravi S. Sandhu (editors): *Second ACM Conference on Data and Application Security and Privacy, CODASPY 2012, San Antonio, TX, USA, February 7-9, 2012*, pages 317–326. ACM, 2012. https://doi.org/10.1145/2133601.2133640. 14

[49] Choudhary, Mahima and Brij Kishore: *Haamd: Hybrid analysis for android malware detection*. In *2018 International Conference on Computer Communication and Informatics (ICCCI)*, pages 1–4. IEEE, 2018. 14

[50] Patel, Zinal D: *Malware detection in android operating system*. In *2018 International Conference on Advances in Computing, Communication Control and Networking (ICACCCN)*, pages 366–370. IEEE, 2018. 14, 15

[51] Liu, Kaijun, Shengwei Xu, Guoai Xu, Miao Zhang, Dawei Sun, and Haifeng Liu: *A review of android malware detection approaches based on machine learning*. IEEE Access, 8:124579–124607, 2020. https://doi.org/10.1109/ACCESS.2020.3006143. 14, 15, 20

[52] Tam, Kimberly, Ali Feizollah, Nor Badrul Anuar, Rosli Salleh, and Lorenzo Cavallaro: *The evolution of android malware and android analysis techniques*. ACM Comput. Surv., 49(4):76:1–76:41, 2017. https://doi.org/10.1145/3017427. 14, 15

[53] Odusami, Modupe, Olusola Abayomi-Alli, Sanjay Misra, Olamilekan Shobayo, Robertas Damasevicius, and Rytis Maskeliunas: *Android malware detection: A survey*. In Florez, Hector, Cesar Diaz, and Jaime Chavarriaga (editors): *Applied Informatics - First International Conference, ICAI 2018, Bogotá, Colombia, November 1-3, 2018, Proceedings*, volume 942 of *Communications in Computer and Information Science*, pages 255–266. Springer, 2018. https://doi.org/10.1007/978-3-030-01535-0_19. 14

[54] Arp, Daniel, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, and Konrad Rieck: *DREBIN: effective and explainable detection of android malware in your pocket*. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*. The Internet Society, 2014. https://www.ndss-symposium.org/ndss2014/drebin-effective-and-explainable-detection-android-malware-your-pocket. 15, 59, 74, 91

[55] Feizollah, Ali, Nor Badrul Anuar, Rosli Salleh, Guillermo Suarez-Tangil, and Steven Furnell: *Androdialysis: Analysis of android intent effectiveness in malware detection*. Comput. Secur., 65:121–134, 2017. https://doi.org/10.1016/j.cose.2016.11.007. 15

[56] Xu, Ke, Yingjiu Li, and Robert H. Deng: *Iccdetector: Icc-based malware detection on android*. IEEE Trans. Inf. Forensics Secur., 11(6):1252–1264, 2016. https://doi.org/10.1109/TIFS.2016.2523912. 15

[57] Kabakus, Abdullah Talha, Ibrahim Alper Dogru, and Aydin Cetin: *APK auditor: Permission-based android malware detection system*. Digit. Investig., 13:1–14, 2015. https://doi.org/10.1016/j.diin.2015.01.001. 15

[58] Alam, Shahid, Zhengyang Qu, Ryan Riley, Yan Chen, and Vaibhav Rastogi: *Droidnative: Automating and optimizing detection of android native code malware variants*. Comput. Secur., 65:230–246, 2017. https://doi.org/10.1016/j.cose.2016.11.011. 15

[59] Bhattacharya, Abhishek and Radha Tamal Goswami: *Dmdam: data mining based detection of android malware*. In *Proceedings of the first international conference on intelligent computing and communication*, pages 187–194. Springer, 2017. 15

[60] Wu, Songyang, Pan Wang, Xun Li, and Yong Zhang: *Effective detection of android malware based on the usage of data flow apis and machine learning*. Inf. Softw. Technol., 75:17–25, 2016. https://doi.org/10.1016/j.infsof.2016.03.004. 15

[61] Sihag, Vikas, Manu Vardhan, and Pradeep Singh: *A survey of android application and malware hardening*. Comput. Sci. Rev., 39:100365, 2021. https://doi.org/10.1016/j.cosrev.2021.100365. 15

[62] Ahvanooey, Milad Taleby, Qianmu Li, Mahdi Rabbani, and Ahmed Raza Rajput: *A survey on smartphones security: Software vulnerabilities, malware, and attacks*. CoRR, abs/2001.09406, 2020. https://arxiv.org/abs/2001.09406. 15

[63] Demontis, Ambra, Marco Melis, Battista Biggio, Davide Maiorca, Daniel Arp, Konrad Rieck, Igino Corona, Giorgio Giacinto, and Fabio Roli: *Yes, machine learning can be more secure! A case study on android malware detection.* IEEE Trans. Dependable Secur. Comput., 16(4):711–724, 2019. https://doi.org/10.1109/TDSC.2017.2700270. 16

[64] Gardiner, Joseph and Shishir Nagaraja: *On the security of machine learning in malware c&c detection: A survey.* ACM Comput. Surv., 49(3):59:1–59:39, 2016. https://doi.org/10.1145/3003816. 16

[65] Li, Li, Alexandre Bartel, Tegawendé F. Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick D. McDaniel: *Iccta: Detecting inter-component privacy leaks in android apps.* In Bertolino, Antonia, Gerardo Canfora, and Sebastian G. Elbaum (editors): *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, pages 280–291. IEEE Computer Society, 2015. https://doi.org/10.1109/ICSE.2015.48. 16

[66] Maass, Michael, Adam Sales, Benjamin Chung, and Joshua Sunshine: *A systematic analysis of the science of sandboxing.* PeerJ Comput. Sci., 2:e43, 2016. https://doi.org/10.7717/peerj-cs.43. 16

[67] Bordoni, Lorenzo, Mauro Conti, and Riccardo Spolaor: *Mirage: Toward a stealthier and modular malware analysis sandbox for android.* In Foley, Simon N., Dieter Gollmann, and Einar Snekkenes (editors): *Computer Security - ESORICS 2017 - 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017, Proceedings, Part I*, volume 10492 of *Lecture Notes in Computer Science*, pages 278–296. Springer, 2017. https://doi.org/10.1007/978-3-319-66402-6_17. 16

[68] Glanz, Leonid, Patrick Müller, Lars Baumgärtner, Michael Reif, Sven Amann, Pauline Anthonysamy, and Mira Mezini: *Hidden in plain sight: Obfuscated strings threatening your privacy.* CoRR, abs/2002.04540, 2020. https://arxiv.org/abs/2002.04540. 17

[69] Whaley, John, Michael C. Martin, and Monica S. Lam: *Automatic extraction of object-oriented component interfaces.* In Frankl, Phyllis G. (editor): *Proceedings of the International Symposium on Software Testing and Analysis, ISSTA 2002, Roma, Italy, July 22-24, 2002*, pages 218–228. ACM, 2002. https://doi.org/10.1145/566172.566212. 18

[70] Ammons, Glenn, Rastislav Bodík, and James R. Larus: *Mining specifications.* In Launchbury, John and John C. Mitchell (editors): *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002*, pages 4–16. ACM, 2002. https://doi.org/10.1145/503272.503275. 18

[71] Li, Yuanchun, Ziyue Yang, Yao Guo, and Xiangqun Chen: *Humanoid: A deep learning-based approach to automated black-box android app testing.* In *34th*

IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019, pages 1070–1073. IEEE, 2019. https://doi.org/10.1109/ASE.2019.00104. 18, 27, 31, 34, 41, 46

[72] Manzil, Hashida Haidros Rahima and S. Manohar Naik: *Detection approaches for android malware: Taxonomy and review analysis.* Expert Syst. Appl., 238(Part F):122255, 2024. https://doi.org/10.1016/j.eswa.2023.122255. 20

[73] Enck, William, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick D. McDaniel, and Anmol Sheth: *Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones.* In Arpaci-Dusseau, Remzi H. and Brad Chen (editors): *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings*, pages 393–407. USENIX Association, 2010. http://www.usenix.org/events/osdi10/tech/full_papers/Enck.pdf. 20, 23

[74] Zhou, Yajin, Zhi Wang, Wu Zhou, and Xuxian Jiang: *Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets.* In *Network and Distributed System Security Symposium*, 2012. https://api.semanticscholar.org/CorpusID:2504550. 20

[75] Yan, Lok Kwong 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. https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/yan. 20

[76] Enck, William, Machigar Ongtang, and Patrick McDaniel: *On lightweight mobile phone application certification.* In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, page 235–245, New York, NY, USA, 2009. Association for Computing Machinery, ISBN 9781605588940. https://doi.org/10.1145/1653662.1653691. 20

[77] Grace, Michael, Yajin Zhou, Qiang Zhang, Shihong Zou, and Xuxian Jiang: *Riskranker: scalable and accurate zero-day android malware detection.* In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, MobiSys '12, page 281–294, New York, NY, USA, 2012. Association for Computing Machinery, ISBN 9781450313018. https://doi.org/10.1145/2307636.2307663. 20

[78] da Costa, Francisco Handrick, Ismael Medeiros, Thales Menezes, João Victor da Silva, Ingrid Lorraine da Silva, Rodrigo Bonifácio, Krishna Narasimhan, and Márcio Ribeiro: *Exploring the use of static and dynamic analysis to improve the performance of the mining sandbox approach for android malware identification.* Journal of Systems and Software, 183:111092, 2022, ISSN 0164-1212. https://www.sciencedirect.com/science/article/pii/S0164121221001898. 20, 82

[79] Wang, Shanshan, Qiben Yan, Zhenxiang Chen, Bo Yang, Chuan Zhao, and Mauro Conti: *Detecting android malware leveraging text semantics of network flows.* IEEE

Trans. Inf. Forensics Secur., 13(5):1096–1109, 2018. https://doi.org/10.1109/TIFS.2017.2771228. 20

[80] Newsome, J., B. Karp, and D. Song: *Polygraph: automatically generating signatures for polymorphic worms*. In *2005 IEEE Symposium on Security and Privacy (SP'05)*, pages 226–241, 2005. 20

[81] Singh, Sumeet, Cristian Estan, George Varghese, and Stefan Savage: *Automated worm fingerprinting*. In *6th Symposium on Operating Systems Design & Implementation (OSDI 04)*, San Francisco, CA, December 2004. USENIX Association. https://www.usenix.org/conference/osdi-04/automated-worm-fingerprinting. 20

[82] Yegneswaran, Vinod, Jonathon T. Giffin, Paul Barford, and Somesh Jha: *An architecture for generating semantic aware signatures*. In *14th USENIX Security Symposium (USENIX Security 05)*, Baltimore, MD, July 2005. USENIX Association. https://www.usenix.org/conference/14th-usenix-security-symposium/architecture-generating-semantic-aware-signatures. 20

[83] Perdisci, Roberto, Wenke Lee, and Nick Feamster: *Behavioral clustering of http-based malware and signature generation using malicious network traces*. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, page 26, USA, 2010. USENIX Association. 20

[84] Nan, Yuhong, Min Yang, Zhemin Yang, Shunfan Zhou, Guofei Gu, and XiaoFeng Wang: *UIPicker: User-Input privacy identification in mobile applications*. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 993–1008, Washington, D.C., August 2015. USENIX Association, ISBN 978-1-939133-11-3. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/nan. 20

[85] Ren, Jingjing, Ashwin Rao, Martina Lindorfer, Arnaud Legout, and David Choffnes: *Recon: Revealing and controlling pii leaks in mobile network traffic*. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '16, page 361–374, New York, NY, USA, 2016. Association for Computing Machinery, ISBN 9781450342698. https://doi.org/10.1145/2906388.2906392. 20

[86] Arora, Anshul, 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. 21, 22, 82

[87] Taylor, Vincent F., Riccardo Spolaor, Mauro Conti, and Ivan Martinovic: *Appscanner: Automatic fingerprinting of smartphone apps from encrypted network traffic*. In *2016 IEEE European Symposium on Security and Privacy (EuroSP)*, pages 439–454, March 2016. 21

[88] Conti, Mauro, Luigi Vincenzo Mancini, Riccardo Spolaor, and Nino Vincenzo Verde: *Analyzing android encrypted network traffic to identify user actions*. IEEE

Transactions on Information Forensics and Security, 11(1):114–125, Jan 2016, ISSN 1556-6021. 21

[89] Ham, Hyo-Sik and Mi-Jung Choi: *Analysis of android malware detection performance using machine learning classifiers*. In *International Conference on Information and Communication Technology Convergence, ICTC 2013, Jeju Island, South Korea, 4-16 October 2013*, pages 490–495. IEEE, 2013. https://doi.org/10.1109/ICTC.2013.6675404. 21, 22

[90] Kurniawan, Harry, Yusep Rosmansyah, and Budiman Dabarsyah: *Android anomaly detection system using machine learning classification*. In *2015 international conference on electrical engineering and informatics (ICEEI)*, pages 288–293. IEEE, 2015. 21, 22, 82

[91] Goodfellow, Ian, Yoshua Bengio, and Aaron Courville: *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org. 21

[92] Martín, Alejandro, Víctor Rodríguez-Fernández, and David Camacho: *CANDYMAN: classifying android malware families by modelling dynamic traces with markov chains*. Eng. Appl. Artif. Intell., 74:121–133, 2018. https://doi.org/10.1016/j.engappai.2018.06.006. 21, 22

[93] Lashkari, Arash Habibi, Andi Fitriah A.Kadir, Hugo Gonzalez, Kenneth Fon Mbah, and Ali A. Ghorbani: *Towards a network-based framework for android malware detection and characterization*. In *2017 15th Annual Conference on Privacy, Security and Trust (PST)*, pages 233–23309, Aug 2017. 21, 22, 82

[94] Feizollah, Ali, Nor Badrul Anuar, Rosli Salleh, Fairuz Amalina, Shahaboddin Shamshirband, *et al.*: *A study of machine learning classifiers for anomaly-based mobile botnet detection*. Malaysian Journal of Computer Science, 26(4):251–265, 2013. 21, 22, 82

[95] Garg, Shree, Sateesh Kumar Peddoju, and Anil Kumar Sarje: *Network-based detection of android malicious apps*. Int. J. Inf. Sec., 16(4):385–400, 2017. https://doi.org/10.1007/s10207-016-0343-z. 21, 22, 82

[96] Dash, Santanu Kumar, Guillermo Suarez-Tangil, Salahuddin J. 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, SP Workshops 2016, San Jose, CA, USA, May 22-26, 2016*, pages 252–261. IEEE Computer Society, 2016. https://doi.org/10.1109/SPW.2016.25. 21, 22

[97] Alzaylaee, Mohammed K., Suleiman Y. Yerima, and Sakir Sezer: *Dynalog: An automated dynamic analysis framework for characterizing android applications*. CoRR, abs/1607.08166, 2016. http://arxiv.org/abs/1607.08166. 21

[98] Enck, William, Machigar Ongtang, and Patrick D. McDaniel: *Understanding android security*. IEEE Secur. Priv., 7(1):50–57, 2009. https://doi.org/10.1109/MSP.2009.26. 22

[99] Pauck, Felix, Eric Bodden, and Heike Wehrheim: *Do android taint analysis tools keep their promises?* In Leavens, Gary T., Alessandro Garcia, and Corina S. Pasareanu (editors): *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, pages 331–341. ACM, 2018. https://doi.org/10.1145/3236024.3236029. 22

[100] Huang, Wei, Yao Dong, Ana Milanova, and Julian Dolby: *Scalable and precise taint analysis for android.* In Young, Michal and Tao Xie (editors): *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*, pages 106–117. ACM, 2015. https://doi.org/10.1145/2771783.2771803. 23

[101] Zhang, Jie, Cong Tian, and Zhenhua Duan: *An efficient approach for taint analysis of android applications.* Comput. Secur., 104:102161, 2021. https://doi.org/10.1016/j.cose.2020.102161. 23

[102] Boyer, Robert S, Bernard Elspas, and Karl N Levitt: *Select—a formal system for testing and debugging programs by symbolic execution.* ACM SigPlan Notices, 10(6):234–245, 1975. 23

[103] King, James C.: *Symbolic execution and program testing.* Commun. ACM, 19(7):385–394, 1976. https://doi.org/10.1145/360248.360252. 23

[104] Anand, Saswat, Alessandro Orso, and Mary Jean Harrold: *Type-dependence analysis and program transformation for symbolic execution.* In Grumberg, Orna and Michael Huth (editors): *Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24 - April 1, 2007, Proceedings*, volume 4424 of *Lecture Notes in Computer Science*, pages 117–133. Springer, 2007. https://doi.org/10.1007/978-3-540-71209-1_11. 25

[105] Gritti, Fabio, Lorenzo Fontana, Eric Gustafson, Fabio Pagani, Andrea Continella, Christopher Kruegel, and Giovanni Vigna: *SYMBION: interleaving symbolic with concrete execution.* In *8th IEEE Conference on Communications and Network Security, CNS 2020, Avignon, France, June 29 - July 1, 2020*, pages 1–10. IEEE, 2020. https://doi.org/10.1109/CNS48642.2020.9162164. 25

[106] Cai, Haipeng and Barbara G Ryder: *A longitudinal study of application structure and behaviors in android.* IEEE Transactions on Software Engineering, 2020. 27

[107] Li, Yuanchun, Ziyue Yang, Yao Guo, and Xiangqun Chen: *A deep learning based approach to automated android app testing.* CoRR, abs/1901.02633, 2019. http://arxiv.org/abs/1901.02633. 27, 33

[108] Bouckaert, Stefan, JVV Gerwen, Ingrid Moerman, Stephen C Phillips, and Jerker Wilander: *Benchmarking computers and computer networks.* EU FIRE White Paper, 2010. 27

[109] Buschmann, Frank, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal: *Pattern-Oriented Software Architecture - Volume 1: A System of Patterns*. Wiley Publishing, 1996, ISBN 0471958697. 29, 39

[110] Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., USA, 1995, ISBN 0201633612. 30, 40

[111] Mao, Ke, Mark Harman, and Yue Jia: *Sapienz: multi-objective automated testing for android applications*. In Zeller, Andreas and Abhik Roychoudhury (editors): *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, pages 94–105. ACM, 2016. https://doi.org/10.1145/2931037.2931054. 31, 33, 34

[112] *Logcat*. https://http://developer.android.com/tools/help/logcat.html. Accessed: 2020-03-15. 31, 40

[113] Allix, Kevin, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon: *Androzoo: collecting millions of android apps for the research community*. In Kim, Miryung, Romain Robbes, and Christian Bird (editors): *Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, Austin, TX, USA, May 14-22, 2016*, pages 468–471. ACM, 2016. https://doi.org/10.1145/2901739.2903508. 31, 41, 61

[114] Su, Ting, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su: *Guided, stochastic model-based GUI testing of android apps*. In Bodden, Eric, Wilhelm Schäfer, Arie van Deursen, and Andrea Zisman (editors): *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, pages 245–256. ACM, 2017. https://doi.org/10.1145/3106237.3106298. 33, 34

[115] *Droidxp source code*. https://github.com/droidxp/benchmark. Accessed: 2020-08-21. 34

[116] *Comscore*. https://www.comscore.com/Insights/Presentations-and-Whitepapers/2018/Global-Digital-Future-in-Focus-2018. 35, 58

[117] *statcounter*. https://gs.statcounter.com/os-market-share/mobile/worldwide. Accessed: 2021-02-10. 35

[118] Shull, Forrest, Jeffrey C. Carver, Sira Vegas, and Natalia Juristo Juzgado: *The role of replications in empirical software engineering*. Empir. Softw. Eng., 13(2):211–218, 2008. https://doi.org/10.1007/s10664-008-9060-1. 36

[119] Zeng, Xia, Dengfeng Li, Wujie Zheng, Fan Xia, Yuetang Deng, Wing Lam, Wei Yang, and Tao Xie: *Automated test input generation for android: are we really there yet in an industrial case?* In Zimmermann, Thomas, Jane Cleland-Huang, and Zhendong Su (editors): *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA,*

November 13-18, 2016, pages 987–992. ACM, 2016. https://doi.org/10.1145/2950290.2983958. 41

[120] Backes, Michael, Sebastian Gerling, Christian Hammer, Matteo Maffei, and Philipp von Styp-Rekowsky: *Appguard - fine-grained policy enforcement for untrusted android applications*. In García-Alfaro, Joaquín, Georgios V. Lioudakis, Nora Cuppens-Boulahia, Simon N. Foley, and William M. Fitzgerald (editors): *Data Privacy Management and Autonomous Spontaneous Security - 8th International Workshop, DPM 2013, and 6th International Workshop, SETOP 2013, Egham, UK, September 12-13, 2013, Revised Selected Papers*, volume 8247 of *Lecture Notes in Computer Science*, pages 213–231. Springer, 2013. https://doi.org/10.1007/978-3-642-54568-9_14. 42, 63, 65

[121] James, Gareth, Daniela Witten, Trevor Hastie, and Robert Tibshirani: *An Introduction to Statistical Learning: With Applications in R*. Springer Publishing Company, Incorporated, 2014, ISBN 1461471370. 43, 45, 66

[122] Rasthofer, Siegfried, Steven Arzt, and Eric Bodden: *A machine-learning approach for classifying and categorizing android sources and sinks*. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*. The Internet Society, 2014. https://www.ndss-symposium.org/ndss2014/machine-learning-approach-classifying-and-categorizing-android-sources-and-sinks 44

[123] Hurier, Médéric, Guillermo Suarez-Tangil, Santanu Kumar Dash, Tegawendé F Bissyandé, Yves Le Traon, Jacques Klein, and Lorenzo Cavallaro: *Euphony: harmonious unification of cacophonous anti-virus vendor labels for android malware*. In *Proceedings of the 14th International Conference on Mining Software Repositories*, pages 425–435. IEEE Press, 2017. 48

[124] Wohlin, Claes, Per Runeson, Martin Hst, Magnus C. Ohlsson, Bjrn Regnell, and Anders Wessln: *Experimentation in Software Engineering*. Springer Publishing Company, Incorporated, 2012, ISBN 3642290434. 55

[125] Silva, Eneias, Alessandro Ferreira Leite, Vander Alves, and Sven Apel: *Expruna : a domain-specific approach for technology-oriented experiments*. Softw. Syst. Model., 19(2):493–526, 2020. https://doi.org/10.1007/s10270-019-00749-6. 55

[126] Arcuri, Andrea and Lionel Briand: *A practical guide for using statistical tests to assess randomized algorithms in software engineering*. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, page 1–10, New York, NY, USA, 2011. Association for Computing Machinery, ISBN 9781450304450. https://doi.org/10.1145/1985793.1985795. 56

[127] 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 Trans. Inf. Forensics Secur., 12(6):1269–1284, 2017. https://doi.org/10.1109/TIFS.2017.2656460. 56

[128] Martin, William J., Federica Sarro, Yue Jia, Yuanyuan Zhang, and Mark Harman: *A survey of app store analysis for software engineering*. IEEE Trans. Software Eng., 43(9):817–847, 2017. https://doi.org/10.1109/TSE.2016.2630689. 58

[129] *Statista*. https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/. Accessed: 2022-02-10. 58, 81

[130] Tam, Kimberly, Ali Feizollah, Nor Badrul Anuar, Rosli Salleh, and Lorenzo Cavallaro: *The evolution of android malware and android analysis techniques*. ACM Comput. Surv., 49(4), jan 2017, ISSN 0360-0300. https://doi.org/10.1145/3017427. 58

[131] Jr., Nataniel P. Borges, Jenny Hotzkow, and Andreas Zeller: *Droidmate-2: a platform for android test generation*. In Huchard, Marianne, Christian Kästner, and Gordon Fraser (editors): *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, pages 916–919. ACM, 2018. https://doi.org/10.1145/3238147.3240479. 59

[132] Rafiq, Husnain, Nauman Aslam, Muhammad Aleem, Biju Issac, and Rizwan Hamid Randhawa: *Andromalpack: enhancing the ml-based malware classification by detection and removal of repacked apps for android systems*. Scientific Reports, 12(1):19534, 2022. 61, 87

[133] Alrawi, Omar, Miuyin Yong Wong, Athanasios Avgetidis, Kevin Valakuzhy, Boladji Vinny Adjibi, Konstantinos Karakatsanis, Mustaque Ahamad, Douglas M. Blough, Fabian Monrose, and Manos Antonakakis: *Sok: An essential guide for using malware sandboxes in security applications: Challenges, pitfalls, and lessons learned*. CoRR, abs/2403.16304, 2024. https://doi.org/10.48550/arXiv.2403.16304. 62

[134] Zhu, Shuofei, Jianjun Shi, Limin Yang, Boqin Qin, Ziyi Zhang, Linhai Song, and Gang Wang: *Measuring and modeling the label dynamics of online anti-malware engines*. In Capkun, Srdjan and Franziska Roesner (editors): *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 2361–2378. USENIX Association, 2020. https://www.usenix.org/conference/usenixsecurity20/presentation/zhu. 62, 67, 77

[135] Rahali, Abir, Arash Habibi Lashkari, Gurdip Kaur, Laya Taheri, François Gagnon, and Frédéric Massicotte: *Didroid: Android malware classification and characterization using deep image learning*. In *ICCNS 2020: The 10th International Conference on Communication and Network Security, Tokyo, Japan, November 27-29, 2020*, pages 70–82. ACM, 2020. https://doi.org/10.1145/3442520.3442522. 62

[136] Sebastián, Silvia and Juan Caballero: *Avclass2: Massive malware tag extraction from AV labels*. In *ACSAC '20: Annual Computer Security Applications Conference, Virtual Event / Austin, TX, USA, 7-11 December, 2020*, pages 42–53. ACM, 2020. https://doi.org/10.1145/3427228.3427261. 62, 92

[137] Li, Li, Tegawendé F. Bissyandé, and Jacques Klein: *Simidroid: Identifying and explaining similarities in android apps*. In *2017 IEEE Trustcom/BigDataSE/ICESS, Sydney, Australia, August 1-4, 2017*, pages 136–143. IEEE Computer Society, 2017. https://doi.org/10.1109/Trustcom/BigDataSE/ICESS.2017.230. 63

[138] *anzhi*. http://www.anzhi.com/. Accessed: 2021-02-15. 63

[139] Spearman, C.: *The proof and measurement of association between two things*. The American Journal of Psychology, 15(1):72–101, 1904, ISSN 00029556. http://www.jstor.org/stable/1412159, visited on 2022-08-09. 66

[140] Falsina, Luca, Yanick Fratantonio, Stefano Zanero, Christopher Kruegel, Giovanni Vigna, and Federico Maggi: *Grab 'n run: Secure and practical dynamic code loading for android applications*. In *Proceedings of the 31st Annual Computer Security Applications Conference, Los Angeles, CA, USA, December 7-11, 2015*, pages 201–210. ACM, 2015. https://doi.org/10.1145/2818000.2818042. 73, 74

[141] Ruggia, Antonio, Andrea Possemato, Savino Dambra, Alessio Merlo, Simone Aonzo, and Davide Balzarotti: *The dark side of native code on android*. Authorea Preprints, 2023. 74

[142] Qian, Chenxiong, Xiapu Luo, Yuru Shao, and Alvin T. S. Chan: *On tracking information flows through JNI in android applications*. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014, Atlanta, GA, USA, June 23-26, 2014*, pages 180–191. IEEE Computer Society, 2014. https://doi.org/10.1109/DSN.2014.30. 74

[143] Felt, Adrienne Porter, Matthew Finifter, Erika Chin, Steve Hanna, and David A. Wagner: *A survey of mobile malware in the wild*. In Jiang, Xuxian, Amiya Bhattacharya, Partha Dasgupta, and William Enck (editors): *SPSM'11, Proceedings of the 1st ACM Workshop Security and Privacy in Smartphones and Mobile Devices, Co-located with CCS 2011, October 17, 2011, Chicago, IL, USA*, pages 3–14. ACM, 2011. https://doi.org/10.1145/2046614.2046618. 81

[144] Surendran, Roopak, Tony Thomas, and Sabu Emmanuel: *Gsdroid: Graph signal based compact feature representation for android malware detection*. Expert Syst. Appl., 159:113581, 2020. https://doi.org/10.1016/j.eswa.2020.113581. 81

[145] Merlo, Alessio, Antonio Ruggia, Luigi Sciolla, and Luca Verderame: *You shall not repackage! demystifying anti-repackaging on android*. Comput. Secur., 103:102181, 2021. https://doi.org/10.1016/j.cose.2021.102181. 81

[146] Costa, Francisco, Ismael Medeiros, Leandro Oliveira, João Calássio, Rodrigo Bonifácio, Krishna Narasimhan, Mira Mezini, and Márcio Ribeiro: *Scaling up: Revisiting mining android sandboxes at scale for malware classification*. arXiv preprint arXiv:2505.09501, 2025. Paper has been accepted to ECOOP'25. 82

[147] Uhlár, Juraj, Martin Holkovic, and Vít Rusnák: *Pcapfunnel: A tool for rapid exploration of packet capture files*. In Banissi, Ebad, Anna Ursyn, Mark W. McK. Bannatyne, João Moura Pires, Nuno Datia, Mao Lin Huang, Weidong Huang,

Quang Vinh Nguyen, Kawa Nazemi, Boris Kovalerchuk, Minoru Nakayama, John Counsell, Andrew Agapiou, Farzad Khosrow-shahi, Hing-Wah Chau, Mengbi Li, Richard Laing, Fatma Bouali, Gilles Venturini, Marco Temperini, and Muhammad Sarfraz (editors): *25th International Conference Information Visualisation, IV 2021, Sydney, Australia, July 5-9, 2021*, pages 69–76. IEEE, 2021. https://doi.org/10.1109/IV53921.2021.00021. 83, 87

[148] Lashkari, Arash Habibi, Gerard Draper-Gil, Mohammad Saiful Islam Mamun, and Ali A. Ghorbani: *Characterization of tor traffic using time based features*. In Mori, Paolo, Steven Furnell, and Olivier Camp (editors): *Proceedings of the 3rd International Conference on Information Systems Security and Privacy, ICISSP 2017, Porto, Portugal, February 19-21, 2017*, pages 253–262. SciTePress, 2017. https://doi.org/10.5220/0006105602530262. 84

[149] Umer, Muhammad Fahad, Muhammad Sher, and Yaxin Bi: *Flow-based intrusion detection: Techniques and challenges*. Comput. Secur., 70:238–254, 2017. https://doi.org/10.1016/j.cose.2017.05.009. 84

[150] Sperotto, Anna, Gregor Schaffrath, Ramin Sadre, Cristian Morariu, Aiko Pras, and Burkhard Stiller: *An overview of IP flow-based intrusion detection*. IEEE Commun. Surv. Tutorials, 12(3):343–356, 2010. https://doi.org/10.1109/SURV.2010.032210.00054. 84

[151] Xie, Donghong: *On the influence of feature selection and regression models on the decoding accuracy of seen and imagery objects using hierarchical visual features*. In *International Conference on Human Machine Interaction, ICHMI 2022, Beijing, China, May 6-8, 2022*, pages 7–15. ACM, 2022. https://doi.org/10.1145/3560470.3560472. 84

[152] Amiriebrahimabadi, Mohammad and Najme Mansouri: *A comprehensive survey of feature selection techniques based on whale optimization algorithm*. Multim. Tools Appl., 83(16):47775–47846, 2024. https://doi.org/10.1007/s11042-023-17329-y. 84

[153] Stephenson, William T.: *Faster and easier: cross-validation and model robustness checks*. PhD thesis, Massachusetts Institute of Technology, USA, 2022. https://hdl.handle.net/1721.1/143247. 86

[154] Awad, Mohammed and Salam Fraihat: *Recursive feature elimination with cross-validation with decision tree: Feature selection method for machine learning-based intrusion detection systems*. J. Sens. Actuator Networks, 12(5):67, 2023. https://doi.org/10.3390/jsan12050067. 86

[155] Zhu, Shuofei, Jianjun Shi, Limin Yang, Boqin Qin, Ziyi Zhang, Linhai Song, and Gang Wang: *Measuring and modeling the label dynamics of online Anti-Malware engines*. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2361–2378. USENIX Association, August 2020, ISBN 978-1-939133-17-5. https://www.usenix.org/conference/usenixsecurity20/presentation/zhu. 86

[156] Wang, Shanshan, Zhenxiang Chen, Qiben Yan, Bo Yang, Lizhi Peng, and Zhongtian Jia: *A mobile malware detection method using behavior features in network traffic.* J. Netw. Comput. Appl., 133:15–25, 2019. https://doi.org/10.1016/j.jnca.2018.12.014. 92, 94

[157] Fallah, Somayyeh and Amir Jalaly Bidgoly: *Android malware detection using network traffic based on sequential deep learning models.* Softw. Pract. Exp., 52(9):1987–2004, 2022. https://doi.org/10.1002/spe.3112. 92

[158] Wang, Peng, Zhijie Tang, and Junfeng Wang: *A novel few-shot malware classification approach for unknown family recognition with multi-prototype modeling.* Comput. Secur., 106:102273, 2021. https://doi.org/10.1016/j.cose.2021.102273. 92

[159] Conti, Mauro, Shubham Khandhar, and Vinod P.: *A few-shot malware classification approach for unknown family recognition using malware feature visualization.* Comput. Secur., 122:102887, 2022. https://doi.org/10.1016/j.cose.2022.102887. 92

[160] Moser, Andreas, Christopher Kruegel, and Engin Kirda: *Limits of static analysis for malware detection.* In *23rd Annual Computer Security Applications Conference (ACSAC 2007), December 10-14, 2007, Miami Beach, Florida, USA*, pages 421–430. IEEE Computer Society, 2007. https://doi.org/10.1109/ACSAC.2007.21. 95

[161] Lv, Zhengwei, Chao Peng, Zhao Zhang, Ting Su, Kai Liu, and Ping Yang: *Fastbot2: Reusable automated model-based gui testing for android enhanced by reinforcement learning.* In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE 2022)*, 2022. 96

[162] Li, Yuanchun, Ziyue Yang, Yao Guo, and Xiangqun Chen: *Humanoid: a deep learning-based approach to automated black-box android app testing.* In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, ASE '19, page 1070–1073. IEEE Press, 2020, ISBN 9781728125084. https://doi.org/10.1109/ASE.2019.00104. 96

Appendix A

Feature Extraction: List of 76 features used in Study 3

Feature Name: Description

Flow duration: Duration of the flow in Microsecond
total Fwd Packet: Total packets in the forward direc.
total Bwd packets: Total packets in the backward direc.
total Length of Fwd Packet Total size of packet in forward direc.
total Length of Bwd Packet Total size of packet in backward direc.
Fwd Packet Length Min: Minimum size of packet in forward direc.
Fwd Packet Length Max: Maximum size of packet in forward direc.
Fwd Packet Length Mean: Mean size of packet in forward direc.
Fwd Packet Length Std: Standard deviation size of packet in forward direc.
Bwd Packet Length Min: Minimum size of packet in backward direc.
Bwd Packet Length Max: Maximum size of packet in backward direc.
Bwd Packet Length Mean: Mean size of packet in backward direc.
Bwd Packet Length Std: Standard deviation size of packet in backward direc.
Flow Bytes/s: Number of flow bytes per second
Flow Packets/s: Number of flow packets per second
Flow IAT Mean: Mean time between two packets sent in the flow
Flow IAT Std: Standard deviation time between two packets sent in the flow
Flow IAT Max: Maximum time between two packets sent in the flow
Flow IAT Min: Minimum time between two packets sent in the flow
Fwd IAT Min: Minimum time between two packets sent in the forward direc.
Fwd IAT Max: Maximum time between two packets sent in the forward direc.
Fwd IAT Mean: Mean time between two packets sent in the forward direc.
Fwd IAT Std: Standard deviation time between two packets sent in the fwd direc.

Fwd IAT Total : Total time between two packets sent in the forward direc.
Bwd IAT Min: Minimum time between two packets sent in the backward direc.
Bwd IAT Max: Maximum time between two packets sent in the backward direc.
Bwd IAT Mean: Mean time between two packets sent in the backward direc.
Bwd IAT Std: Standard deviation time between two packets sent in the bwd direc.
Bwd IAT Total: Total time between two packets sent in the backward direc.
Fwd PSH flags: Times the PSH flag was set in pkt travelling in the fwd direc.(0 for UDP)
Bwd PSH Flags: Times the PSH flag was in pkt travelling in the bwd direc.(0 for UDP)
Fwd URG Flags: Times the URG flag was in pkts travelling in the fwd direc.(0 for UDP)
Bwd URG Flags: Times the URG flag was in pkts travelling in the bwd direc.(0 for UDP)
Fwd Header Length: Total bytes used for headers in the forward direc.
Bwd Header Length: Total bytes used for headers in the backward direc.
FWD Packets/s: Number of forward packets per second
Bwd Packets/s: Number of backward packets per second
Packet Length Min: Minimum length of a packet
Packet Length Max: Maximum length of a packet
Packet Length Mean: Mean length of a packet
Packet Length Std: Standard deviation length of a packet
Packet Length Variance Variance length of a packet
FIN Flag Count: Number of packets with FIN
SYN Flag Count: Number of packets with SYN
RST Flag Count: Number of packets with RST
PSH Flag Count: Number of packets with PUSH
ACK Flag Count: Number of packets with ACK
URG Flag Count: Number of packets with URG
CWR Flag Count: Number of packets with CWR
ECE Flag Count: Number of packets with ECE
down/Up Ratio: Download and upload ratio
Average Packet Size: Average size of packet
Fwd Segment Size Avg: Average size observed in the forward direc.
Bwd Segment Size Avg: Average size observed in the backward direc.
Fwd Bytes/Bulk Avg: Average number of bytes bulk rate in the forward direc.
Fwd Packet/Bulk Avg: Average number of packets bulk rate in the forward direc.
Fwd Bulk Rate Avg: Average number of bulk rate in the forward direc.
Bwd Bytes/Bulk Avg: Average number of bytes bulk rate in the backward direc.
Bwd Packet/Bulk Avg: Average number of packets bulk rate in the backward direc.
Bwd Bulk Rate Avg: Average number of bulk rate in the backward direc.
Subflow Fwd Packets: The average number of packets in a sub flow in the fwd direc.
Subflow Fwd Bytes: The average number of bytes in a sub flow in the fwd direc.
Subflow Bwd Packets: The average number of packets in a sub flow in the bck direc.

Subflow Bwd Bytes: The average number of bytes in a sub flow in the backward direc.
Fwd Init Win bytes: The total number of bytes sent in initial window in the fwd direc.
Bwd Init Win bytes: The total number of bytes sent in initial window in the bck direc.
Fwd Act Data Pkts: Count of pkt with at least 1 byte of TCP data payload in the fwd direc.
Fwd Seg Size Min: Minimum segment size observed in the forward direc.
Active Min: Minimum time a flow was active before becoming idle
Active Mean: Mean time a flow was active before becoming idle
Active Max: Maximum time a flow was active before becoming idle
Active Std: Standard deviation time a flow was active before becoming idle
Idle Min: Minimum time a flow was idle before becoming active
Idle Mean: Mean time a flow was idle before becoming active
Idle Max: Maximum time a flow was idle before becoming active
Idle Std: Standard deviation time a flow was idle before becoming active