Preventing the release of illegitimate applications on mobile markets

Lavoisier Wapet

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par :

Institut National Polytechnique de Toulouse (Toulouse INP)

Discipline ou spécialité :

Informatique et Télécommunication

Présentée et soutenue par :

M. PATRICK WAPET

le lundi 5 avril 2021

Titre :

Preventing the release of illegitimate applications on mobile markets

---

Ecole doctorale :

Mathématiques, Informatique, Télécommunications de Toulouse (MITT)

Unité de recherche :

Institut de Recherche en Informatique de Toulouse (IRIT)

Directeurs de Thèse :

M. DANIEL HAGIMONT

M. ALAIN TCHANA

Rapporteurs :

Mme SONIA BEN MOKHTAR, CNRS

M. ROMAIN ROUVOY, Université de Lille

Membres du jury :

M. YEROM DAVID BROMBERG, UNIVERSITE RENNES 1, Président

M. ALAIN TCHANA, ECOLE NORMALE SUP LYON ENS DE LYON, Membre

M. DANIEL HAGIMONT, TOULOUSE INP, Membre

Acknowledgements

My thanks go first of all to my Creator. In addition to the grace He gives me every day, He has allowed me to complete this thesis. May He continue to enlighten me and allow me to share His joy and His serenity with my relatives, and with His family that I hope will grow in His Son, the First Born.

Then I thank my parents, Blaise and Antoinette and my brothers, sisters an even extended family Isabelle, Elisabeth, Albert, Ange and Maurice.You have always supported me unfailingly, allowing me to carry out with great serenity my early and long studies up to this Ph.D. I am very grateful to you.

I would like to thank my thesis supervisors, Daniel Hagimont and Alain Tchana for their effective advice and supervision. I also thank them because they gave me the opportunity to start and complete a thesis on the other side of the world, thus exploring many faraway lands from my home country Cameroon;

I also want to express all my gratitude to Mr Romain Rouvoy and Ms Sonia Ben Mokhtar who agreed to be the reporters of my thesis, as well as to all the jury members.

Thanks of course to our secretaries Sylvie Armengaud, Muriel Pernier and Annabelle Sansus, without whom our research would be much more difficult.

Many thanks to SCALE team at the I3S laboratory in Sophia Antipolis, I'm thinking of Fabrice, Andrea, Alessio. Thank you to the members of the laboratory of the AVALON team of the ENS of Lyon; here I think of Christian, Eddy, and my desk mate Laurent. Thanks also to Andre Ole Ravnas who accepted to introduce me to mobile phone hacking and dynamic instrumentation. His advices were very useful. I would also like to thank Louison and Mohamed who, in addition to their technical support, helped me to persevere in the delicate context that marked last years of this thesis.

Finally, I would like to thank all my colleagues from the IRIT laboratory, starting with Boris Teabe, who has always encouraged me in his own somewhat severe way, when necessary. I also thank Vlad Nitu whom I admire for his frankness, without forgetting Gregroire Todeschi and Fopa Leon Constantin who, with Boris Teabe and Eric Munier, were part of my welcoming committee at the laboratory. Thanks also to Brice Ekane for his simplicity and wisdom;

Of course, I can't forget Stella Ndonga Bitchebe and Mvondo Djob the rocket with whom we defended each other when we had to adapt during the

many trips we made during the thesis. and then Kevin Jiokeng , Tu Ngoc, Bao Bui, Mathieu Bacou the producer of inspiring templates, Peterson , Armel and especially Kouam Josiane, and Boris Wouembe, God knows why for these last two, mainly.

Thank you all for your respective roles in my life as a Ph.D. student and in my thesis!

Abstract

The popularity of mobile applications has been growing worldwide over the last few decades. This popularity is attracting more and more authors of malicious applications called malwares. To detect those malwares, mobile markets have implemented analysis methods that suffer from several limitations. Those we have identified and which we propose to solve in the scope of this thesis are mainly two . The first is the inability to cope with a new method of malware publication consisting in anticipating the mobile version of a company that does not yet have one. The second limitation is the difficulty encountered by dynamic analysis solutions to be able to scale. To solve the first limitation we designed and implemented a security check system called IMAD (Illegitimate Mobile App Detector), which is based mainly on online search engines and machine learning techniques. To solve the second problem, we introduced a scalable tracing approach, that we call *delegated instrumentation*. It leverages Android's instrumentation module and mainly relies on ART (Android Runtime) reverse engineering and hacking. The evaluation results show that IMAD can protect companies from anticipation attacks with an acceptable error rate and at a low cost for MMPs. And we demonstrated the effectiveness of the *delegated instrumentation* with a prototype named Odile that traces various app types (including benign apps and malware) on Samsung Galaxy A7 2017. In particular, we show how much Odile outperforms Frida, the state-of-the-art tool in the domain.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In order to speed up data processing, Information Technology has become established first in institutions through computers, then in homes through workstations and finally in the pockets or close to the hands of modern man through mobile phone on which mobile applications (or mobile apps) are installed.

Mobile apps obviously facilitate most human activities. They do so by giving them access to social networks that are becoming increasingly virtual, by allowing them to consume online business services or by providing them with multimedia content. Their implementation requires an ecosystem in which the main producer is the application developer. The latter submits its application to the Mobile Application Markets, which in turn make it available for download on users' personal phones.

As mobile apps are at the heart of transactions and may often manipulate data belonging to users who increasingly trust them, hackers have begun to build malicious application, mainly to harm and hijack these critical transactions and steal sensitive data available on mobile phones. In order to do this, hackers must first find a way to deploy their malicious mobile apps on their targets' phones, and then execute their malicious loads by exploiting the internal architecture of mobile apps and mobile operating system. Hackers must also update their malware to ensure that it evolves and is future-proofed despite the countermeasure techniques that are also increasingly emerging.

In the light of all these threats, countermeasures have been proposed in the literature. They fall into two categories, although solutions combining both categories exist. These are static methods on the one hand and dynamic methods on the other. Static methods simply examine the application package and its components, while dynamic methods analyse all the exploitable data during mobile app runtime . All of these methods, in addition to being accurate, must not only be able to evolve to counter the new sophisticated approaches implemented by hackers, but must also be easily integrated into the mobile application ecosystem and more specifically into the deployment process without compromising scalability. It is precisely to provide new solutions to the last two issues that the work contained in this thesis has been carried out.

The first contribution we present identifies and then proposes a countermeasure against a new malicious mobile apps deployment strategy, i.e. a new method that hackers use to get their mobile apps installed on users' mobile phones. This new approach consists of passing their application not as another known application, but as the mobile version of a known Internet service that does not yet have one. It thus bypasses existing detection systems based on known malware signature comparisons. Our contribution, on the other hand, explores the service and company catalogues available online as well as search engines to find out whether the graphic charts of mobile apps newly submitted for publication do not use the above-mentioned technique. To this exploration are coupled the data-mining and optimization algorithms that make it effective. At the end of this procedure the imitated services are found and alerted.

The second contribution of our thesis is a tool for tracing mobile apps. It aims to solve a problem faced by new dynamic analysis solutions requiring the intervention of users who are unfortunately not very qualified to collect data on their phones. By dispensing with the "rooting" of the mobile phone, and by maintaining an acceptable level of accuracy, this proposed tool brings a new breath to reboost the crowd-source malware analysis. It is based on indirect code injection, a technique that subtlely reuses components already present on the operating system, thus facilitating the tracing of most java calls from a mobile app.

1.1 Publications that constitute this thesis:

The first work of this thesis has been published at FGCS under the reference.

- Patrick Lavoisier Wapet, Alain-Bouzaïde Tchana, Tran Giang Son, Daniel Hagimont. Preventing the propagation of a new kind of illegitimate apps. Future Generation Computer Systems, Elsevier, 2019, 94, pp.368-380. (10.1016/j.future.2018.11.051). (hal-02495523)

The second contribution of this thesis is named *Odile* and has been published at COMPAS, presented at the GDR 2020 and is currently being submitted to the conference

- Patrick Lavoisier Wapet, Alain-Bouzaïde Tchana, Louison Gitzinger, Daniel Hagimont, David Bromberg. Odile: A scalable tracing system for non-rooted and on the shelf Android devices Systems.

1.2 Thesis organization

Chapter 2 first offers a panoramic view of the thesis context, namely the ecosystem of mobile applications. It then presents threats to this ecosystem and provides an overview of possible approaches to address these threats. Then it describes issues faced by the latter approaches and conclude on issues targetet by this thesis contributions.

Chapters 3 and 4 present the contributions of this thesis. For each of them, they present the specific context and motivation, the design, implementation details, evaluation, and finally a brief overview of the state of the art. Chapter 3 identifies a new way of deploying malicious applications and presents a countermeasure to deal with it. Chapter 4 proposes a solution to the scalability problem of crowd-based dynamic analysis solutions. This solution consist of a new mobile app tracing tool adapted to non-expert users and implementing a technique called the *delegated instrumentation*.

Finally, chapter 5 concludes these developments, and suggests a vision for the future of security in the mobile applications ecosystem.

Chapter 2

Background and contributions

We first describe in this chapter the context of the thesis, namely the ecosystems of mobile applications (or mobile app(s)). Then, we list the threats faced by users of the latter mobile apps and detail the existing approaches that tries to mitigate them. We conclude this chapter with a brief overview of our main contributions.

2.1 Mobile app Ecosystem

We first highlight mobile app usefulness in everyday life. Then we describe the typical path the latter app before ending up on end user's mobile phone. Then, we list his components, what they do and how they interact. We also list interaction between the latter mobile app and and other ones on the one hand, and between the mobile app and the operating system on the other hand. We end by presenting Android, the operating system used by one of *the three dominant mobile app marketplaces* [53].

2.1.1 Adoption of mobile apps in human society

Mobile apps have gradually become an integral part of everyday life, as two-thirds of the world's inhabitants use them[12]. These are software applications that run on mobile phones and allow people to perform several common tasks such as phone calls, messaging, web browsing, education, social network chatting, multimedia and video games. the table 2.1 gives an overview of the most popular features in the mobile markets for popular mobile apps such as Android Google Play Store, Apple Store, and Windows Market.

Mobile apps functionalities	**Examples**	**Percentage on Apple Store**	**Percentage on Google Play Store**	**Percentage on Windows Phone Market Place**
Games	Pokemon Go, Angry Bird	18%	13%	14%
Health	Home Workout, Step Tracker	4%	3%	3%
Books	wikiHow, Kindle	11%	9%	10%
Music	Deezer, Spotify	5%	6%	2%
News	CNN, Franceinfo,	6%	7%	11%
Business	Linkedin, Remote Desktop	14%	14%	18%
Travel	Oui.sncf, trivago	8%	6%	6%
Social	Facebook, Instagram	2%	5%	3%
Finance	PayPal	2%	2%	2%
Education	EDX, Math Tests	10%	5%	6%
Entertainment	Netflix, Arte	10%	11%	19%
Lifestyle	MakeMeBetter, RelaxRain	8%	8%	5%
Photography	Adobe Photoshop	2%	1%	2%
Personnalization	Animals Ringtones, Themes	-	10%	-

Table 2.1: table of the most used app functionalities (from [53])

Mobile apps, and the choice of Android Operating Systems: A mobile app is strongly associated with the operating system on which it is supposed to run. Thus there are mobile apps for Android, iOS and Symbian OS. Among these systems, we focus on the Android system since it is the most popular and more than 52% of phones sold are equipped with it [2]. Moreover it is possible to contribute to its improvement: its source code (under the name of Android Open Source Project) is built on more than 100 open source projects including the Linux kernel. However, there are similarities between the different mobile operating systems. They therefore encounter more or less the same problems. So the contributions we propose mainly for the Android system may well apply to other systems.

Mobile Apps developement An Android mobile app comes as a package called *APK* (Android Application Package). It contains code in DEX format and other files that act as resources for the application. To produce such a package, the developer writes java code from the android SDK, and or C/C++ in the *AndroidStudio* development environment, and then associates resources such as logos, images, graphical interfaces as XML files. Afterwards, he compiles and signs the resulting package with his private key using the *apkSigner* tool available in *Androidstudio*.

Mobile Apps publication As soon as the mobile app is ready to be installed on an Android phone, the developer must make it public. To do this, as shown in the figure 2.1, he submits it to the mobile app market after opening a developer account there. The mobile market takes the newly submitted mobile app through a security analysis phase before making it downloadable by any android phone user worldwide.

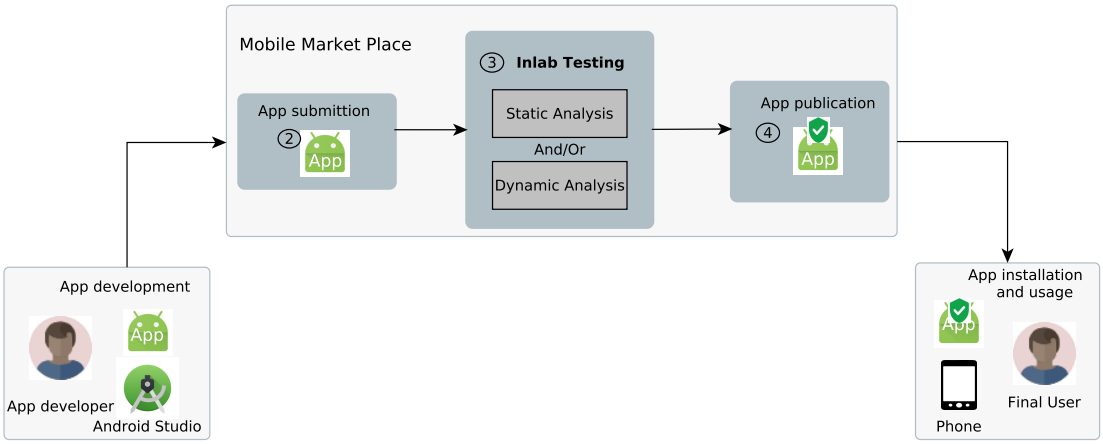


Figure 2.1: **Typical mobile App publication process.**

In the following, we see what a mobile application looks like and what it is made of.

2.1.2 Mobile app structure

A mobile app comes in the form of an archive, commonly called APK (Android Application Package). This archive presented in the figure 2.2 contains a file called **AndroidManifest.xml** that the operating system uses to know the name of the APK, the phone resources it will have access to and the different operations and services it will execute during its operation. In the following, we give a little more detail on the key information contained in the **AndroidManifest.xml**.

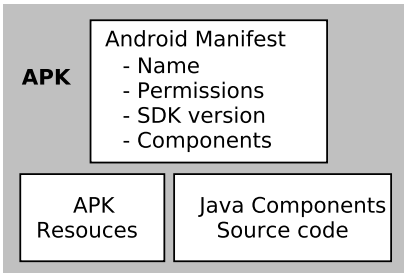


Figure 2.2: **Overview of the Android Package (APK) content .**

- **APK name:** This is a package name that the developer assigns to his mobile app. Ideally, this name should be unique in the world of Android apps. Thus, by using this name, it is possible for the system to identify a specific app.
- **Permission:** To be able to use certain resources and functionalities of the phone, the developer of the mobile app must request the permissions associated with them from the operating system. It does so by declaring these permissions in the **AndroidManifest.xml**. As an example, permission READ_PHONE_STATE, allows the app to read the phone state information like device id, which otherwise is forbidden. Other permissions allow the mobile app to receive and send *communication intents*, the main communication means between mobile apps installed on the same phone and between the same mobile apps components.
- **SDK Version:** Mobile app is mainly developed in java language, strongly coupled with the Android SDK (Software Development kit). This SDK provides a set of APIs used by the developer to access functionalities offered by the operating system. As these functionalities are constantly

evolving through Android versions [47], the developer must mention in the **AndroidManifest.xml** file the versions most compatible with his mobile app. He does this by defining the $minSdkVersion$, $maxSdkVersion$ and $targetSdkVersion$ attributes. Note that it is also possible to integrate C/C++ code into a mobile application by using the JNI (Java Native Interface), which bridges the gap between C implementations and the JAVA calls and vice versa.

- **Components:** The developer organizes the source code of his mobile app around the main classes called components that he extends from the SDK. These classes, which must be declared in the **AndroidManifest.xml** file, govern the life cycle and the interactions between the java objects of the mobile app. Thus, the visible objects associated with graphical interfaces and with which the user interacts are called **Activity**. During the interactions between the user and the mobile app, some tasks can be executed in the background by **Services**. Some activities and services are activated when events external to the mobile app occur such as the phone's waking up, in this case the developer must declare **BroadcastReceiver** that listen to these events before activating the concerned activities and services. Finally, all of the above components handle data that the developer organizes and provides them through the definition of **Content Providers**.

In addition to the AndroidManifest.xml file, the APK contains other **resources** specific to the mobile app such as images (the logo for example), graphical interfaces described in XML format also called layout, character strings used by the mobile app and many others files like native libraries.

Finally, the APK contains **mobile app code** compiled in DEX(Dalvik Executable) format. This source code will be interpreted by a DALVIK/ART virtual machine that is part of the operating system at runtime.

2.1.3 The Android Operating System

A brief history of the Android system : Android, the system software that is avalaible for phones, tablets, and more generally for any kinds of smart devices, was first developed by a company called Android Inc. In the early 2000's. This company has been acquired by Google in 2005 and the latter made a first public release of the Android System in 2008, named Android 1.0.

Since then, Google has released over 20 Android versions, on average every six months, to include new features and address new hardware requirements. In practice, each release of Android is referred to by multiple names: (1) its

version number (e.g., Android 4.4); (2) its API level (e.g., API level 19); and (3) a name of sweet (e.g., KitKat). The API level is even used to determine whether a mobile app is compatible with an Android SDK version prior to installing the mobile app on the device.

Android software stack description: When downloaded and installed on the mobile phone, the mobile app interacts with the operating system according to the software stack shown in figure 2.3. Let us describe it from the bottom up:

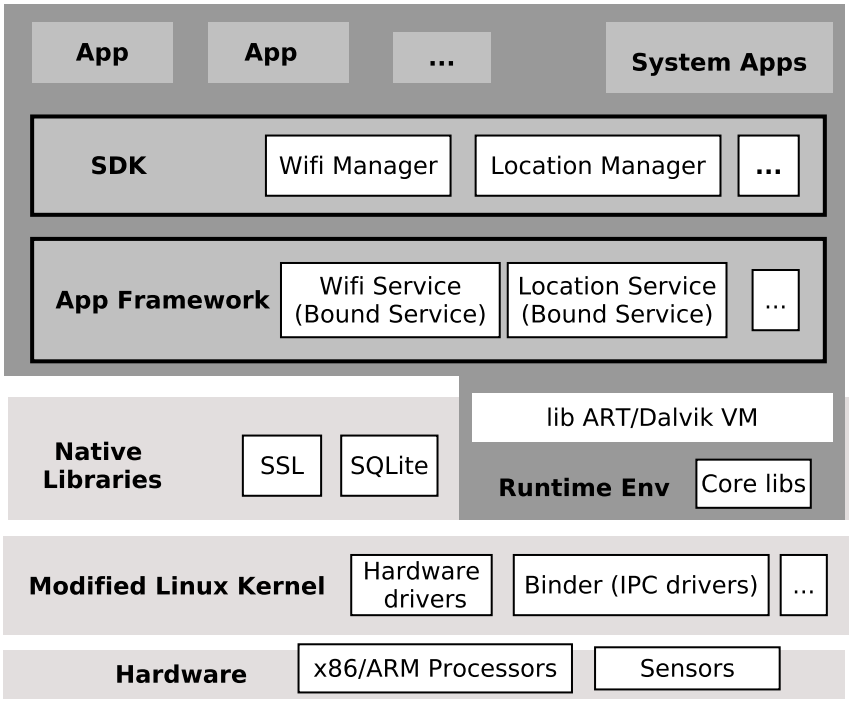


Figure 2.3: **Overview of the Android Operating System (OS) Architecture.**

- Android is built on a **Linux kernel 2.6**, enriched with some drivers such as Binder IPC for inter-process communication, or other drivers such as the gyroscope driver for sensor controlling.
- Above this kernel are **libraries**, some of which extend the drivers defined above. These libraries constitute the native layer of the operating system. They include the Dalvik/ART virtual machine that executes the DEX or Native code of each mobile app in a process having its own linux rights, and coming from the cloning of the Zygote process, the first process from the operating system.

- The source code of a mobile app, in order to access the system's functionalities, consumes, as we have said, Java APIs whose implementation is carried out in the **framework** layer. This layer, also executed by the Dalvik/ART virtual machine, contains system services developed in Java, such as the LocationManager which supports Android apps retrieving GPS coordinates of a device.
- In the last layer we find all mobile apps including **system apps** such as Home, Contact, etc. that are usually shipped with Android devices, and **user apps** such as Facebook, Twitter. that can be installed by users on-demand form Android markets (e.g., from Google Play, the official market maintained by Google).

After presenting the key concepts associated with the ecosystem of mobile apps from their design to their functioning, we discuss in the following section the different threats that affect their users.

2.2 Major threats to mobile app users

Malicious individuals or hackers are at the origin of attacks against the Android ecosystem, that's why to better understand these attacks, we cite the malicious intentions that can animate the hackers and then we cite the steps usually followed by the latter in order to put their plan into action.

2.2.1 Common hacker intentions

By providing useful functionalities to phone users, mobile apps manipulate valuable information and are at the heart of multiple, sometimes monetary transactions. It is mainly to **harm users**, **steal their data** and **hijack the mechanisms** governing the often critical operations they carry out with their phones that hackers have been interested in mobile apps.

User nuisance with or without payment (Randsomewares, denial of service)

Sometimes hackers want to harm the peacefulness of the users in exchange for payment or for vain glory.
For example, there are mobile apps called Randsomewares that, once installed, force the operating system to maintain a static and unchangeable image on the screen that becomes insensitive to manipulation, making the phone unusable. To unlock the infected phone, the owner must make a bank transfer, whose procedure is usually displayed in the image that appears. In 2015–2016, two

big mobile ransomware families, Small and Fusob [13] represent more than 93% of the mobile ransomware space.

Similarly, sabotage attacks such as denial of service can be orchestrated on phones as it is already the case on websites.

Sensitive Data stealing

Data manipulated by mobile apps is the subject of envy by hackers because it can be sold on spammers' sites, belonging to the dark web or directly reused for malicious purposes. Thus a hacker who has access to a secret password can usurp the identity of the owner and use the mobile services to which he has access without his knowledge. It can be his mailbox, his bank account, his social networks. Similarly, hackers use data from mobile apps to profile their users. They do this after having subverted, for example, their purchase history, their travel history, their messaging content and the main theme of their favourite multimedia content. The profiling carried out in this way is most often used for targeted advertising and espionage, among other things.
Mobile apps manipulate data from multiple sources:

- **The user** who, through sensors such as touch screen, GPS, microphone, camera, giroscope, provides data to the mobile apps or operating system. In addition, by activating wifi, GSM or bluetooth connections, the user again, gives them access to data such as phone conversations or network packets.
- The **internal storage** of the phone, decomposed into three parts, the private data to each mobile app installed on the phone, the data available on the external cards, and external data that an app can download from remote servers.
- **Data exchange channels** between mobile apps, the storage system, and source sensors.

Hackers will implement all possible tricks in their mobile apps, interacting with the operating system and the user in order to control the information sources listed above and thus steal certain data. For example, once installed the malware will display the graphical interface of a mobile app known as facebook. And the user, after entering his credentials through the touch keyboard, will have them stolen.

There is another example: as Bring Your Own Devices (BYODs) become popular in enterprise environment, an attacker could also use mobile malware

to exploit and access a victim's private network. Once the victim's network is compromised, the attacker could access corporate resources, steal corporate data.

Internal mobile app task abuse.

Sometimes hackers, in order to obtain direct financial gains, exploit the workflow of commercial services as long as these services can integrate mobile apps.

Some online shops advertise on blogs. On these blogs there are advertising spaces on which they publish the link to their site. Each time a user of the blog clicks on this link, the owner of the blog is paid by the shop site. So the more clicks the blog has, the more income the owner gets. Hackers who own blogs can use a mobile app to automatically generate user clicks from a mobile phone and make a lot of money.

Similarly, some mobile services can be purchased using airtime credits. For example, when paying the bill for a ringtone, the subscriber has to send an SMS to a premium-rate number and then receives another confirmation SMS to confirm the purchase. The subscriber is billed after confirming the second SMS. However, if a hacker succeeds in having this second validation done by a stealth mobile app that he has managed to have installed on the subscriber's phone, he will make money that he will receive from the mobile phone agency on the account and without the subscriber's knowledge.

2.2.2 Key attack implementation steps

In order to carry out their attacks and sustain them over time, hackers typically proceed in three steps, as shown in figure 2.4, the first step consist in finding a way to get their mobile apps to their target's phone. The second step is executing the threat itself. The third step is maintaining their malware so that they can bypass detection techniques that are improving by the day.

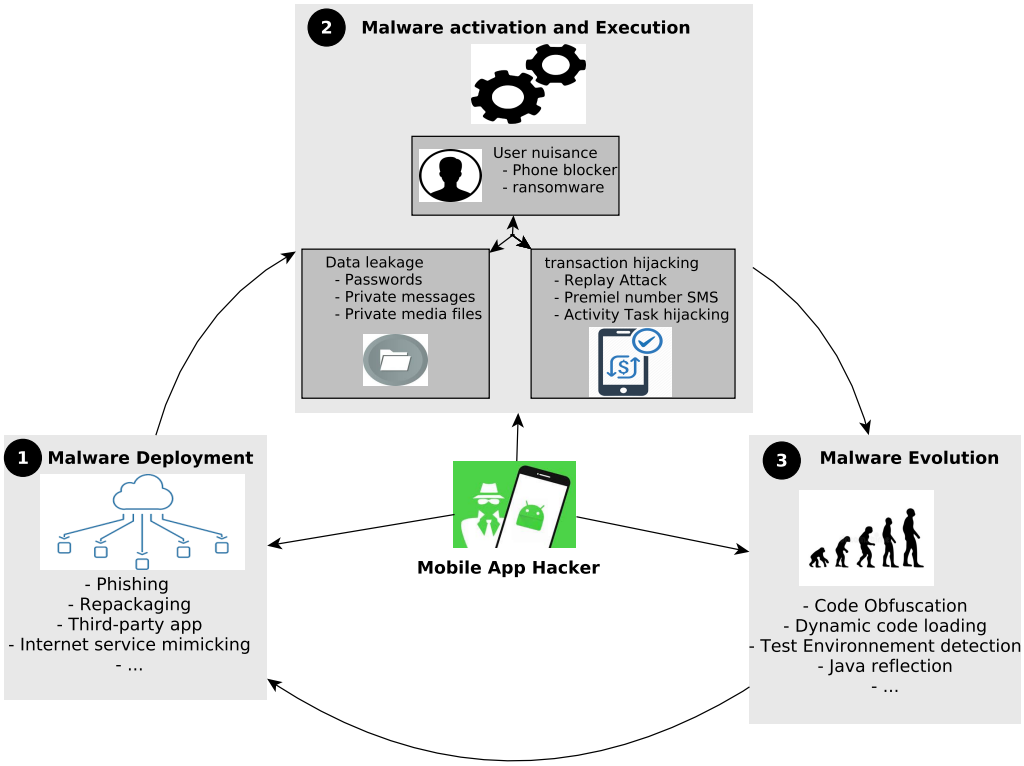


Figure 2.4: **Overview of different mobile apps hackers tasks.**

Malware deployment

To bypass the normal distribution channel for mobile apps, hackers typically use phishing as a subterfuge, in which the user installs the mobile apps from a source other than the official mobile market. They may also repackage a well known mobile app after hiding their malicious code in it. The objective being that the targeted user installs the repackaged mobile app believing it to be the original one. The attacker can also copy the appearance (name, icon and sometimes even GUI) of another known mobile app in order to impersonate it and have it installed.

Malware execution

Actions carried out during this step depend strongly on the hacker final intention (we discussed possible intentions in section 2.2.1), and on OS architecture. Thus in his code the hacker can perform the following actions:

- Increase the scope of possible actions that can be carried out by their mo-

bile app once they are installed on the phone. They do this by requesting the maximum possible permissions. For example [45] proves that with the permissions SYSTEM_ALERT_WINDOW and the BIND_ACCESSIBILITY_SERVICE an app can completely control the UI feedback loop and create devastating attacks.

- They can implement by their function call flow, some means of conveying information from a data source (such as a file or a text input field) to an output such as (another file or an internet server) without the user's knowledge. This practice is motivated by the data stealing.
- Reuse methods or malicious libraries or APIs to hijack critical and exploitable transactions such as monetary transactions, or advertising. The motivation of this practice is to hijack transactions.
- Use some Framework APIs to block the screen or encrypt user data. These actions **are mainly carried out by randsomewares**. This practice may be motivated by the desire to harm the user.

Malware evolution

This evolution consists for the hacker to update his malware to override the new detection methods deployed on mobile market places. Hackers who see their malware removed from the ecosystem will be willing to implement subterfuges and then come back and stay as long as possible. So they will insert into their malware **camouflage methods such as code obfuscation, dynamic loading, runtime environment checking** [104]

2.3 Mobile malware detection methods

The main weapon that hackers use to foment attacks within the mobile ecosystem malware analysis techniques is the mobile app itself. That is why malwares analysis techniques rely on the latter, that is **mobile app** to highlight either **malicious intent** or **one of an attack's implementation steps**, or combinations of both. To highlight these **malicious behaviors**, as we will call them later, the mobile app can be analyzed either statically, meaning based on the APK, or dynamically, meaning while it is running.

2.3.1 Static analysis

Static analysis consists of analyzing the APK of the mobile app without running it. Static analysis techniques are generally classified according to the part of the APK (described previously in 2.1.2) that is analyzed.

Analysis based on the file *AndroidManifest.xml*

Static analysis approaches that use this file manly focus on permissions that are declared within it. Many frameworks, such as Pscout [22] and Whyper [69], use static analysis to evaluate the risks of the Android permission system on individual apps. It has also been shown by others tudies [77], that malware requests more permissions than benign apps. In the million apps Andrubis (an example of malware analysis tool [57]) received from 2010 to 2014, malicious apps requested, on average, 12.99 permissions, while benign apps asked for an averageof 4.5.

Some analysis systems like Drebin [19] target the intents that the mobile app can receive, also declared in the AndroidManifest file. In one scenario, private data can be leaked to a malicious app that requested the data through intents defined as receivable in its Android manifest file.

The analysis may also focus on the components the app has declared because a malware that executes in background often has a service component and a receiver component in order to receive the boot intent on system booting. Throughchecking components and their received intents, analyst may have a brief view of the potential behaviour of a mobile app. As an example in Droid-Mat [88] , intents, permissions, component deployment, and APIs were extracted from the Manifest and analyzed with several machine-learning algorithms, such as k-means, k-nearest neighbors, and naive Bayes, to develop malware detection systems.

Resource-based analysis

Malware detection approaches that rely on mobile app resources do so primarily to detect repackaging. Indeed, since resources include the XML description of a mobile app's graphical interfaces and the images or files they use, an attacker who wants to reuse these elements to repackage a mobile app is forced to reuse some of its resources. Based on this observation, some systems like [73], proposes an algorithm for comparing mobile app resources based on data-mining in order to detect cases of repackaging, which are booming the mobile market (86% of hackers used it in 2014 to deploy their malware).

dexFile-based analysis

The files *.dex*, difficult for humans to read, are obtained by compiling the mobile app java source code. The static analysis solutions based on these

files, aim primarily at extracting and analyzing features such as classes, APIs, methods, method call sequences, and program dependency graphs (CFGs). To extract the features contained in the *.dex* files it is sometimes useful to use tools such as Dex2Jar, undex, JEB, Dexpler, Androiguard, dedex, Pegasus [20] in order to decompile them in a more readable format adapted to the analysis solution used. There are many levels of formats, from low level bytecode or assembly code to human-readable source code. In general, more drastic decompiling methods have a higher fail rate or error rate, due to the significant change from the old format to the new one, some of which can be amended by post-processing.

From the decompiled format, features (e.g., classes, APIs, methods), structure sequences, and program dependency graphs can be extracted and analyzed using machine learning algorithms [52], much more advanced techniques rely on decompiled dexfiles to track down the flow of intents in interprocesscommunications (IPC), also known as inter-component communications (ICC) [96] , and to aid smart stimulation [61].

APK-hybrid analysis

Finally, some analysis approaches such as DREBIN [19] exploit all parts of the mobile app at the same time by collecting intents, permissions, app components, APIs, and network addresses from malicious APKs, with a detection rate that can reach 94 % of the malware and with low false-positive rate.

Limits of static analysis

Static analysis is criticized for being powerless when hackers use hiding techniques such as code obfuscation or dynamic loading from a remote server for example, in order to hide the malicious load that will run on the mobile phone once the mobile app is installed. In this case this malicious code is not visible in the APK at the time of static analysis. Hence the need to use dynamic analysis. We will discuss about this in the next section.

2.3.2 Dynamic Analysis

Purpose and test environment. The main purpose of dynamic analysis is to obtain from mobile apps additional data that are not accessible by static analysis, thus dynamic analysis first consists in launching the mobile app to be tested in a test environment. In the field of dynamic analysis, this environment

can be an emulator such as the android virtual device or simply a mobile phone; even as of these two tools the emulator is the most used because it is much more accessible. In any case, the test environment have to be equipped with a tool that can collect the required data.

Starting and handling. Once the mobile app is started in the test environment, an utilisation scenario is performed on it in order to test several app execution paths. The author of the simulation can be a bot (e.g. Google's android Monkey Tool), a tool from the research [23] or more simply a human as the current trend [48] suggests. The objective during this simulation being to increase the chances to ativate malicious actions, the more precise and complete the manipulation is, thas is covering the largest number of crutial execution path, the more conclusive the analysis will be.

Data capture and analysis. Finally, data collected from sensors installed on the test environment are collected for analysis. Typically, anything related to the threats we've listed above (see section 2.2.1) is investigated. More precisely, by capturing for example the graphical interfaces produced by the system such as [62], one could look for illicit deployment intentions such as copying the graphical interfaces for phishing or repackaging. Also by observing the android Framework APIs that the mobile app uses as well as its way of using certain libraries (advertising for example), we will be able to detect the willingness of malware (APIs to block the screen) or hijacking of user transactions. Finally, by observing the obfuscation APIs, such as those for java reflection, or dynamic loading of native code, or detection of the test environment (because some malware does not execute if it is started on an emulator), one can then suspect a willingness to hide and thus deepen the search on the suspected App.

Dynamic analysis classification. Dynamic analysis techniques are generally classified according to the architecture set up to examine the mobile app. Some approaches collect data at an existing layer of the Android system, while others rely on virtualization, adding additional layers. The latter only work with emulators. The State of the Art section of the third chapter describes some of them. Still, the further you go from the mobile app layer, the less human readable the information collected is [46]. For example, when capturing system calls it is impossible to know the name of the java API that is at the origin of a call. Hence the need to infer from the data obtained for further information

Limits of dynamic analysis

Although dynamic methods of analysis are very useful, and give good results, they present several limitations: firstly, the bots used in the handling

phase are not very accurate in covering the execution paths, secondly, when humans are involved in the test process, many constraints (such as the need to have rooted phones, the reduced number of experts for the test) make the dynamic approach less scalable.
Sometime static analysis are combined with dynamic analysis. By combining these two approaches, we achieve hybrid analysis, the main topic of the next sub-section.

2.3.3 Hybrid Analysis

Hybrid analysis methods overcome some limitations of static and dynamic analysis when used separately. The main advantages of hybrid analysis include the following::

- The possibility to predict in advance by a static analysis the execution paths to be explored during the dynamic analysis. SmartDroid [101], EvoDroid [61].
- The possibility of checking whether there is consistency between the actions declared in the manifest of the mobile app, which is obtained statically, and the actions carried out by the mobile app during its operation which are obtained dynamically.
- Multi-step detection to filter the most risky mobile apps statically before analyzing them dynamically.
- Iterative detection, which consists of modifying the mobile app statically to insert hooks that allow you to discover certain information dynamically. And if possible to iterate the procedure as did Backes et al. [25] and [87]

2.4 Challenges of mobile malware analysis systems

In this section, we describe the issues facing malware detection systems. Indeed, in order to be fully operational, they must be **accurate** in detecting mobile malwares and **easily deployable** in the Android ecosystem presented in the 2.1.

2.4.1 Accuracy

For the ecosystem of mobile apps to work in a secure way, it is necessary to reduce as much as possible the number of malwares that can be installed by users; at the same time, any benign developer must be able to publish and

have his mobile app installed by his future users without his mobile app being rejected. Finding a compromise between these two previous objectives requires an accurate malware detection solution. This accuracy depends on several elements. For static analysis solutions, these elements are

- The origin of collected data on mobile app (which parts of the APK have been used, which metadata -such as the developer name or the comments on the mobile app page on Google Play- have been exploited,...)
- The Intermediate transformations that these data have undergone (e.g. for the construction of dependency graphs, the organization of selected features)
- The algorithms applied to these representations (machine learning, comparisons..).

For dynamics analysis solutions, these elements are

- The quality of the data collected dynamically (e.g. can we dynamically have the exact name of the API used by the mobile app, and its arguments? Do we just have to be limited only to system calls)?
- The algorithms applied to this data (info-flow comparison, machine learning...).

The third factor that influences the detection solution accuracy is its ability to fight against new deployment techniques, malicious load execution, and concealment that malware integrates to perfect itself. All these improvements depends on the android ecosystem evolution.

2.4.2 Deployability

In addition to being accurate as we have shown previously, malware detection solutions must not interfere with the process of publishing mobile apps to the mobile markets, their functioning on phones, or the operating system life-cycle. For this reason, solutions must:

- Be compatible with several versions of operating systems, several hardware architectures, require at least the least modification of the operating system (mainly in the case of solutions that require the phone not to be rooted).
- Not tamper with the mobile app to the point of disrupting the user experience or allow malware to detect if it has been tampered with. The

latter is worth mentioning here because malware, when modified, does not deploy its malicious load, as hybrid scanning systems often tamper with the mobile app being scanned in order to insert hooks.
- Be scalable by executing in a short time: this is crucial if we want to keep up with the current rate of mobile app publishing on mobile marketplaces, which is more than three thousands every day Google Play [3].

Some of these conditions are not always met by mobile malware detection systems. Hence the contributions we propose in this thesis.

2.5 Contributions

The first contribution that we propose in this thesis is classified among the static analysis solutions. It consists in detecting a new type of malware deployment not yet explored at the current state of our knowledge. These are malicious mobile apps masquerading as mobile versions of web services that do not yet have known mobile apps.

We propose to detect malwares that use this deployment technique by extracting features such as names and logos and then using them to search public web services directories. Once the results obtained from internet, data-mining algorithms are applied to extract the one that is closest to the mobile app to be analyzed and that represents a company. If such a company exists it is contacted for further verification. The mobile app is considered illegitimate if it turns out that its developer has no connection with the company found. This work has been published at FGCS under the reference.

- Patrick Lavoisier Wapet, Alain-Bouzaïde Tchana, Tran Giang Son, Daniel Hagimont. Preventing the propagation of a new kind of illegitimate apps. Future Generation Computer Systems, Elsevier, 2019, 94, pp.368-380. (10.1016/j.future.2018.11.051). (hal-02495523)

The second contribution of this thesis ranks among the solutions of dynamic analysis. Indeed, it is a tool for tracing mobile apps. Its particularity is that it is the very first to allow the tracing of a mobile app on a non-rooted phone in a precise way and by providing as much information as possible. This tool, named Odile, facilitates the democratization of dynamic analysis methods and opens the voice to a new trend in mobile app detection, the crowd-based one. This work has been published at COMPAS 2019, presented at GDR 2020.

- Patrick Lavoisier Wapet, Alain-Bouzaïde Tchana, Louison Gitzinger, Daniel Hagimont, David Bromberg. Odile: A scalable tracing system for non-rooted and on the shelf Android devices Systems.

Chapter 3

Preventing the propagation of a new kind of illegitimate apps

3.1 Abstract

A significant amount of apps submitted to mobile market places (MMP) are illegitimate, resulting in a negative publicity for these MMPs. To cope with this situation, several app scan solutions have been proposed and integrated into MMPs (e.g. Bouncer at Google). To our knowledge, all scanning solutions in this domain only focus on the detection of illegitimate apps which mimic existing ones. However, recent attack analysis reveal the appearance of a new category of victims: enterprises which did not yet publish their app on the MMP. Thereby, an attacker may be one step ahead and publish a malicious app using the graphic identity of a trusted enterprise. Famous enterprises such as Blackberry, Netflix, and Niantic (Pokemon Go) have been subject of such attacks. We designed and implemented a security check system called IMAD (Illegitimate Mobile App Detector) which is able to limit aforementioned attacks. The evaluation results, realized on up to 5, 000 apps/enterprises[1], show that IMAD can protect both big and small and medium-sized companies from such attacks with an acceptable error rate (almost nil on legitimate apps and less than 20% on illegitimate apps). The evaluation results also show that our system is able to check all apps deployed within a day on Google Play or Apple App Store for a cost of around $1,755. The evaluation results show that IMAD can protect companies from such attacks with an acceptable error rate and at a

[1]This number is estimated according to the size of app repository and to companies apps names dedicated to evaluations

low cost for MMPs.

3.2 Introduction

Nowadays, the vast majority of mobile apps are made available to users through digital distribution platforms called mobile market places (MMP) such as Google Play for Android and App Store for Apple. These main MMPs host a tremendous amount of apps. For example, Google Play has more than 3.3 million [18] apps and over 50 billion downloads [91].This plethora of applications in mobile markets drives developers to implement new and honest ways to gain visibility, such as advertising.

Some hackers, on the other hand, in order to increase the visibility of their application and **thus encourage downloading**, associate them with the graphic charter of certain well-known public apps, thus taking advantage of their reputation. A study from a security company [82] revealed that around 77% of the most downloaded apps have at least one illegitimate version. To cope with them, several app scan solutions have been investigated and integrated into MMPs (e.g. Bouncer at Google). To our knowledge, scanning solutions for illegitimate apps only focus on the detection of apps which mimic **existing ones**. However, recent attack analysis reveal the arrival of a new victim category. Attackers develop and publish mobile apps for well-known companies **which did not publish a mobile app yet.** In 2013, the Blackberry messenger has been a victim of such an attack where an illegitimate app has been published on Google Play and downloaded about 100k times before its removal. More recently (September 14, 2016), Pokemon Go has also been attacked in the same way; see the following post [83]:

"*A few days ago we reported to Google the existence of a new malicious app in the Google Play Store. The Trojan presented itself as the "Guide for Pokemon Go". According to the Google Play Store it has been downloaded more than 500,000 times... Kaspersky Lab products detect the Trojan as HEUR:Trojan.AndroidOS.Ztorg.ad. At least one other version of this particular app was available through Google Play in July 2016.*".

The French Telecommunication company Orange has announced the development of a mobile money service called Orange Bank [68] which will be available throughout a mobile app this summer. We have successfully experimented the publication of an illegitimate version of that app on a popular MMP and several downloads have been observed[2].

[2]This experiment was validated by the ethics commission from our laboratory.

This chapter presents IMAD (Illegitimate Mobile App Detector), a solution for detecting this new category of illegitimate apps at submission time (when the developer uploads the app in the MMP). To our knowledge, this is the first research work which investigates this issue. The main principle is to identify, from visible characteristics of the app (e.g., name or logo), the trusted entity (e.g., a company) associated with these characteristics. This trusted entity is either the submitter or the one the submitter wants to mimic. This identified entity is then contacted by email to validate the app submission.

The implementation of IMAD raises several challenges. The most important among them is the following: how to identify the trusted entity associated with an app regarding the number of worldwide trusted entities? IMAD answers this question by relying on the biggest database in the world which is Google (its search engine). Our basic idea is to combine several standard text and image similarity checking in order to find from the internet the legitimate and trusted entity behind each submitted app (its visible characteristics). Although this idea appears simple to label, its implementation is not easy. The evaluation of IMAD with more than 5,000 apps (from AndroZoo [16] and Contagio [36], among other datasets) demonstrates the effectiveness of our approach, with an acceptable margin of error: almost nil on legitimate apps and less than 20% on illegitimate apps. Overall, we made the following contributions:

(1) We highlighted a new security problem which affects all MMPs: the apps presented under the image of a well-known public entity which does not have a mobile app yet.
(2) We presented an algorithm which is able to successfully detect and prevent the above problematic situation. We provide IMAD, a prototype which is easy to exploit and to integrate with existing MMPs.
(3) We evaluated IMAD with more than 5,000 apps, covering all enterprise categories (geographical location, activity, etc.). The evaluation results show that IMAD can protect both big, small and medium-sized companies from such attacks. In addition, our system is able to validate all apps deployed within a day on Google Play or Apple App Store for a minimal cost (about $1,755).
(4) We compared IMAD with existing solutions (namely Androguard [17] and FsQuadra [99]) which confirmed that the studied issue cannot be addressed using current approaches. We showed that IMAD is also able to detect illegitimate situations handled by existing solutions.

The rest of this chapter is organized as follows. A review of the related work is presented in Section 4.7. Section 3.4 defines the concepts used in this

chapter. It also presents the motivations. Section 4.5 presents our contributions while Section 4.6 presents evaluation results. Finally, we present our conclusion in Section 4.8.

3.3 Related work

Many studies contributed to the issue of detecting illegitimate apps. The proposed approaches can be classified according to three main criteria:
When: the **time** when detection is performed. Detection can be performed statically: an analysis of the app's installation files. It can also be performed dynamically when the app is launched. Usually all are realize on MMP, statically it is done on the MMP servers and dynamically it is done on emulator or dedicated phones. But nowadays some part of the dynamic analysis are starting to be deployed on user phones.
How: the employed detection **method**. We classify these methods into two groups: those which attempt to detect **internal** abnormal or suspicious characteristics within the app (e.g., abnormal communications), and those which attempt to detect fake apps through **similarities** with legitimate apps (e.g., based on images or logos).

These criteria logically lead (according to the place/time/method) to the following classes of solutions:

- **static/internal**. Solutions in this class rely on the analysis of app installation files. An example is described in [29] where they analyse the control flow in the app code in order to detect malicious behaviours. Another example is the Android bouncer [67] when looking on electronic signatures.
- **static/similarities**. Solutions in this class aim at detecting similarities between suspicious apps (fake apps) and legitimate apps already published in the MMP. Such similarities may be detected from document files (text, images) packed with the app [100] or from its code [51, 98, 33].
- **dynamic/internal**. In this class, solutions rely on a dynamic analysis, i.e., they execute the app before effectively publishing it in the MMP. This execution is a means to observe the internal behaviour of the app when launched on a device or on an emulator. For instance in [31], they observe runtime communications in order to detect connections with malicious sites. Another example is [89] which verifies that the name of the app (captured from its graphical user interface at runtime) is consistent with the communication endpoints (URLs) used by the app. Also, several

solutions [29, 65, 93] introduce indicators (for instance an image) chosen by users when a (known) legitimate app is installed. If an app imitates a legitimate app without presenting the indicator (in its GUI), the user knows the app is illegitimate.

- **dynamic/similarities**. In this class, solutions are looking for similarities with existing apps in the MMP to detect fake apps, but dynamically before publishing. In [75], they analyse at runtime apps' GUI (inside the MMP) in order to classify apps and detect similarities.

All these contributions aim at detecting illegitimate apps before their publication. The detection may be performed statically by analysing the installation files of the app or dynamically by observing the app's behaviour. As presented in table 3.1 The detection either identifies a malicious behaviour within the app or identifies a similarity with a legitimate app. Detecting malicious behaviours within apps is limited because it is difficult to cover all attacks (and avoid false negatives). Similarity detection appears to be more promising.

Our solution falls into this latter category. However, it does not rely on the detection of pre-identified characteristics (e.g., from the GUI) from already published apps, which would limit its coverage. It detects all attacks, including those targeting apps which do not yet have a mobile version published in an MMP.

How	Similarities	Internal	IMAD
Independent of store data	✓	✗	✓
Independent of store applications	✗	✓	✓
Independent of the location in the world	✗	✗	✓
Independent of the attack mecanism	✓	✗	✓

Table 3.1: Drawbacks and advantages of related work solutions compared to IMAD

3.4 Definitions and Motivations

3.4.1 Definitions

Trusted entity. We define a trusted entity as an enterprise or an institution which is recognised by a government authority (generally through a unique identification number). In this thesis, we are interested in apps which belong to trusted entities, not to private holders. Let's call $\mathcal{T}$ the set of existing trusted entities.

Graphic identity (noted GI). The GI of a trusted entity T (respectively an app A) is the set of visual elements which refer to T (respectively A) without confusion. Most of the time, the GI of an app is included within the GI of the trusted entity which possesses the app. The GI of a trusted entity may be its name, logo or any image which refers to one of its services.

Trusted developer and the attacker. Let us consider an app A, implemented by a developer D. The latter is said to be a trusted developer if the trusted entity T which is behind the GI of A recognises D, otherwise D is considered to be an attacker. Knowing that in some cases (which are extremely rare because entities generally try to define GI so that they are unique) several entities may have similar GI, we consider that the trusted entity behind a GI is the most popular one.

Legitimate and illegitimate app (respectively noted L and I). An app is said to be legitimate if it has been published by a trusted developer, otherwise the app is illegitimate. Notice that the illegitimacy of an app is independent from the developer intent. It means that an illegitimate app is not necessarily dangerous from the user point of view. However, it is from the point of view of the trusted entity because it is steering its users. For instance the Orange Cache Cleaner app (a small tool used for clearing application cached files) could be considered illegitimate because its GI refers to the famous enterprise Orange (the French telecommunication company), especially its service Orange Cash. From this definition, one can easily understand why legitimate/illegitimate app detection systems mainly rely on GI analysis.

Malware and Safe app (respectively noted M and S). An app is said to be a malware if it does actions without the initial approbation of the user, otherwise the app is said to be safe. Subsequently, malware detection

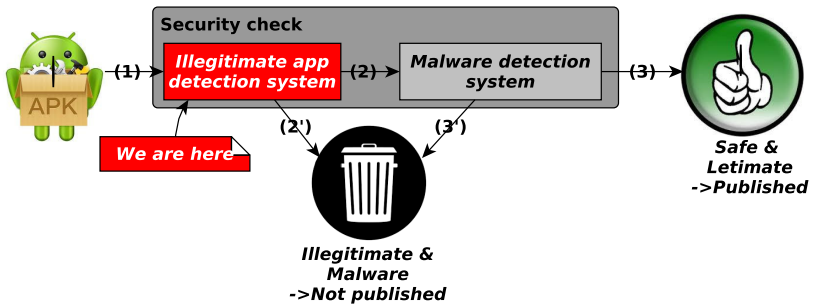


Figure 3.1: Synthetic app submission workflow, from the security check point for view

systems mainly study the *behaviour* of the app.

3.4.2 Research scope

According to the above definitions, apps can be classified into four categories[3]: $L \cap S$, $L \cap M$, $I \cap S$, and $I \cap M$. Except the former category, all the others consist of what we call bad apps (towards the user or a trusted entity). MMP operators try to avoid the publication of bad apps on their platforms. Therefore, before being published, each submitted app is subject to several security checks that can be synthesized in two steps (see Fig. 3.1): GI analysis (for detecting illegitimate apps) and behaviour checking (for detecting malware). Apps which fail one of these controls are kept within a dedicated storage for further studies (e.g. for improving the detection systems). An app is published only if it satisfies all the security checks. In this chapter, we only focus on the detection of illegitimate apps, which are the basis of phishing attacks [31]. Thus, malware detection is out of scope for this chapter. The next section summarizes the current state of the research in this domain in order to highlight our specific contribution. For illustration, we consider Android apps, although our contribution can be applied to other app types.

3.4.3 Problematic

Let us note $\mathcal{A}$ the set of downloadable/published apps in the MMP. Let us note A_{pub} an app under the submission process. Existing illegitimate app detection solutions can be organized into three classes:

(1) source code analysis [29, 98]: they check if there is an app in $\mathcal{A}$ whose

[3]For instance, $L \cap S$ means the intersection of Legitimate and Safe apps.

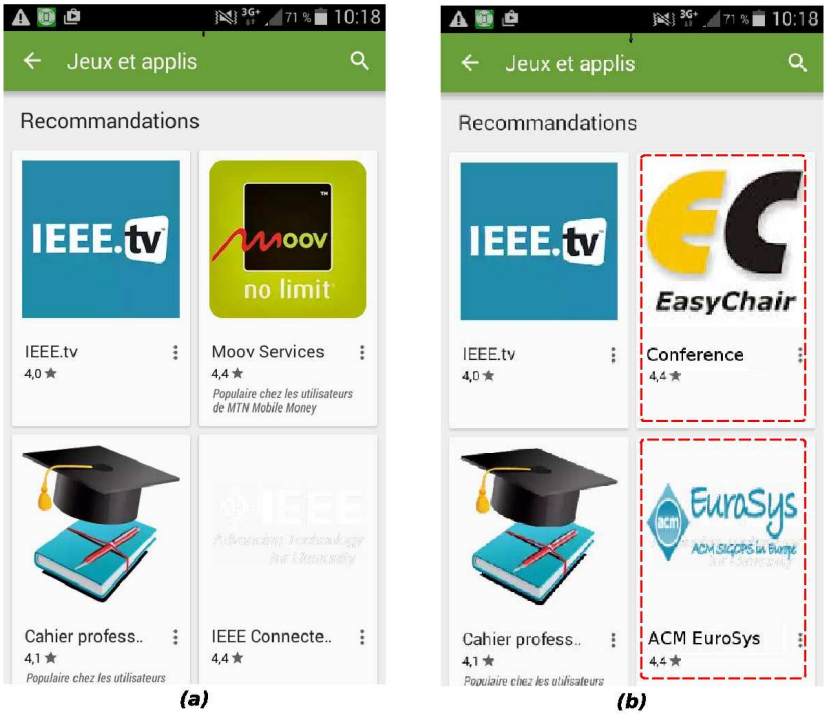


Figure 3.2: Example of attack

implementation structure (especially its graphical user interface) is similar to the one of A_{pub}.

(2) image analysis [100, 33]: they check if there is an app in $\mathcal{A}$ which uses the similar images as A_{pub}.

(3) and app name analysis [65]: they check if there is an app in $\mathcal{A}$ whose name is similar to A_{pub}.

As we can see, all these solutions only focus on detecting if A_{pub} is similar to **an existing app**. More formally, they answer the following question: ***(Q1)*** $\exists A \in \mathcal{A}$ ***(within the MMP), such that*** A_{pub}***'s GI is close to A's GI?***

We claim that answering (Q1) does not allow to cover all illegitimate apps at submission time. Consider the situation where the attacker implements A_{pub} as a service of a trusted entity **which has not yet published** a mobile version of its service. The fraudulent nature of A_{pub} will not be detected by current solutions. Fig. 3.2.a presents the list of apps suggested by Google play when the user is looking for an IEEE app. This suggestion list could have been the one presented in Fig. 3.2.b, which includes two illegitimate apps: EasyChair (faking the legitimate conference management system EasyChair [42]) and ACM EuroSys (faking the EuroSys conference management system). This

situation may occur in current MMPs because the legitimate organization behind EasyChair for example has not yet published a mobile app version of the system[4]. For example, the attacker could obtain the *username* and the *password* of the conference reviewers. Therefore, their reviews could be the subject to Man-in-the-Middle attacks. Another illegitimate situation we experimented concerns the french telecommunication company Orange. The latter has announced the arrival of a mobile money app called Orange Bank [68] this year. Our team has developed and successfully published on a popular MMP[5] an app which purports to be Orange Bank. This situation could have been very problematic for the legitimate company and its clients in the case of a real attacker. Notice that we have unpublished the app after one month so as to avoid exposure to legal proceedings

Considering the large success of smartphones combined with the trend of converting computer applications to mobile apps, this problem is crucial. Therefore, illegitimate app detection systems should not limit their checks to the GIs within the MMP. More formally, instead of answering (Q1) as current researchers do, we answer the following question: ***(Q2)*** $\exists T \in \mathcal{T}$ ***(worldwide), such that*** A_{pub}***'s GI is close to T's GI?*** This is a very difficult problem. This work presents (for the first time) a solution to this issue.

3.5 IMAD: Illegitimate Mobile App Detector

3.5.1 Overall System Design

This chapter presents IMAD, a solution for detecting illegitimate apps at publication time. Its works as follows, summarized in Fig. 3.3. Once the app is uploaded, IMAD builds its GI, noted GI_A_{pub}. Then it performs a set of web searches (on the internet), analyzed with standard machine learning techniques using each element in GI_A_{pub} in order to find the trusted entity behind GI_A_{pub}. This is the core of our solution. We assume that any trusted entity can be found on the web[6]. If the result of the previous step reveals the presence of at least one trusted entity (noted T), a validation email is sent to it and a countdown is armed. The app is considered to be illegitimate if IMAD

[4]Notice that it totally makes sense to have the mobile version of these apps. It would be useful for conference organizers.

[5]The name of the MMP is not revealed in order to avoid negative publicity.

[6]One may ask why not just getting the contact of T from the submitter in order to accelerate searches. This would not be secure because our system does not trust the developer, who could be an attacker.

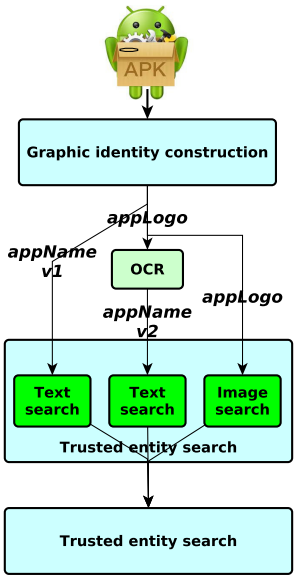


Figure 3.3: IMAD general functioning

does not receive a validation email before the end of the countdown. Notice that if the developer is legitimate, thus the trusted entity will waiting for the validation email sent by IMAD.

IMAD can be deployed in two manners: directly within a specific MMP or deployed as an independent service (IMAD as a Service or IMADaaS). We consider this latter case because it is the most generic one. Therefore, once an MMP is registered as an IMADaaS customer, it can automatically forward all received apk (Android Package Kit - we mainly experimented with Android apps) to IMADaaS for checking. Upon receiving the checking result, the MMP can apply its internal checking system (see Fig. 3.1).

The implementation of IMAD raises several challenges. The most important among them are: (1) Trusted entity determination: This challenge complements the challenges associated with the fight against mobile malware that we listed in the background section of this thesis. The reason is that in addition to the mobile application ecosystem, we are also interested in public entities that may be web services or companies that have built up a reputation. So **how to cover all trusted entities which exist throughout the world? How to obtain their GI, knowing that there is no database which includes them?** (2) GI comparison: This challenge is related to the accuracy of our approach. **How to identify, with as less errors as possible, the proximity between GIs? How to minimize both the false positive**

and false negative rates? (3) Scalability: This challenge is related to the deployability of our approach. **the time, the amount of resources as well as the cost** required for exploiting IMAD should be acceptable. The next sections details each IMAD's component while tackling the above challenges. To facilitate reading, difficult concepts are introduced (when needed) and followed by illustrations. The latter are based on the illegitimate EasyChair app presented in the previous section.

3.5.2 Graphic identity (GI) construction

IMAD considers the following elements as part of GI_A_{pub}: the name of the app (noted $appName$) and its logo. These elements are chosen because they are those which mainly influence the user's decision in the process of associating an app with a trusted entity. Concerning the name, we consider two versions: the one given by the developer (called *appName v1* in Fig. 3.3), and the other overlaid onto the logo (called *appName v2* in Fig. 3.3). This second version is important because sometimes the attacker can provide a bizarre name, knowing that the relevant one is well visible on the logo. The extraction of all these elements is straightforward. We use *apktool* [35] to extract both *appName v1* and the logo from the apk. Then we[7] use tesseract-ocr [74], an optical character reader system (OCR in Fig. 3.3), to extract *appName v2*. Applied to our illustrative example, *appName v1* could be "BXdFcGKfp1" while *appName v2* is "EC EasyChair" (refer to the logo in Fig. 3.2.b).

3.5.3 Trusted entity search

After the construction of the submitted app's GI (noted GI_A_{pub}), IMAD has to find the trusted entity (if exists) which is behind the name or the logo of the app. Our basic idea is to rely on the web (especially the Google custom search engine) in order to consider all trusted entities. In fact, we assume that attackers only target trusted entities which are known by a significant number of persons, and we are betting that such entities are indexed by Google. Each element of GI_A_{pub} is used as a search criteria, all searches being performed in parallel. Therefore, we distinguish two search types:

- text search: performed on *appName v1* and *appName v2*, see Section 3.5.4 to Section 3.5.9.
- image search: performed on the logo, see Section 3.5.10. This step exploits almost the same algorithms as the text search.

[7] "we" is used to designate IMAD

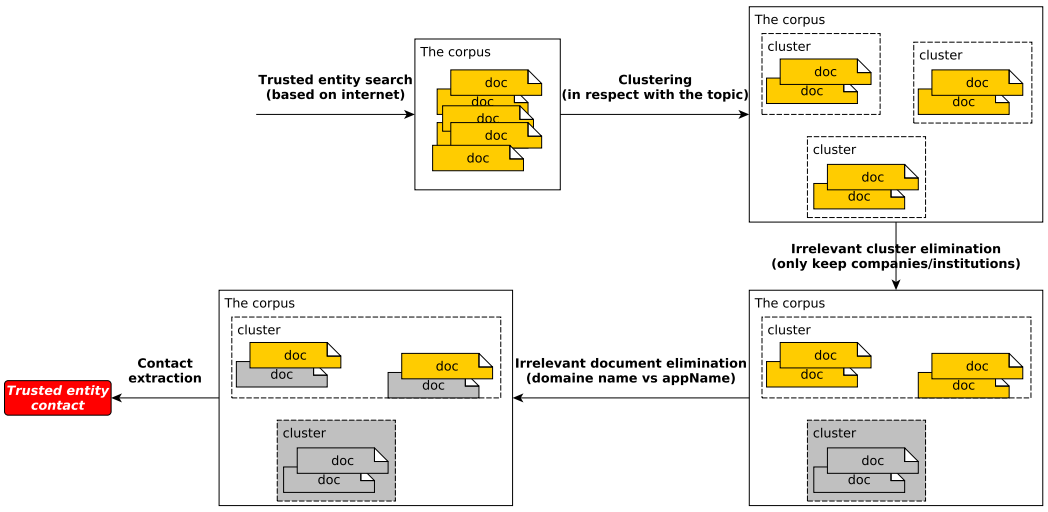


Figure 3.4: The main steps of our solution

3.5.4 Text Search (based on $appName$)

The main difficulty is to filter from the search results all pages which are not directly related to the official website of the trusted entity behind $appName$. To this end, we use a five-step algorithm, summarized in Fig. 3.4:

- (1) collection of web pages which relate to $appName$;
- (2) organization of web pages into clusters according to the topic they deal with;
- (3) elimination of clusters whose topic does not relate to a trusted entity, the rest are merged;
- (4) elimination of pages which are not directly related to $appName$ (non official pages, youtube pages, press articles, etc.);
- (5) extraction of the contact (email) of the trusted entity.

3.5.5 Web page collection

We rely on the Google Custom Search framework [49] to perform this task. Google's APIs can be used for programming custom Google searches. The result of a search is a list of items which represent web pages. Each returned item is composed (among others) of a title, a brief description of the web page, and a link to its HTML content. In the case of IMAD, we have experimentally seen that the first 20 items are sufficient (see the evaluation section). From each item, IMAD builds what we call a *document* (noted *doc*) by concatenating the name, the description and the HTML content. Each document then goes

Corpus built from *appName v1*="BXdFcGKfp1" search results

```
No document!
```

Corpus built from *appName v2*="EC EasyChair" search results

```
doc1 http://easychair.org/
doc2 https://en.wikipedia.org/wiki/EasyChair
doc3 http://voronkov.com/easychair.cgi
doc4 http://www.magisdesign.com/fr/elenco_prodotti/easy-family/easy-chair/
doc5 http://www.ikea.com/gb/en/products/sofas-armchairs/armchairs/nolmyra-↘
    easy-chair-birch-veneer-grey-art-10233532/
doc6 https://headgum.com/the-easy-chair
```

Figure 3.5: Illustration of the Web page collection step

through few changes (such as conversion to lower-case, elimination of HTML tags and stop words, etc.) in order to facilitate the next steps. The obtained set of documents is called the *corpus*. Notice that documents which contain very few information (such as 404 pages) are removed from the corpus. Fig. 3.5 presents the corpus built from our illustrative example.

3.5.6 Clustering

Intuitively, clustering consists in gathering documents from the *corpus* which nearly have the same group of words (the application of this step to our illustrative example is presented in Fig. 3.6). To this end, we used the k-means clustering algorithm. K-means works on vectors while we deal with a corpus. Several studies have investigated the issue of corpus vectorization. IMAD uses Vector Space Model [54], a widely used solution.

$Cluster_1$

```
doc1 http://easychair.org/
doc2 https://en.wikipedia.org/wiki/EasyChair
doc3 http://voronkov.com/easychair.cgi
```

$Cluster_2$

```
doc4 http://www.magisdesign.com/fr/elenco_prodotti/easy-family/easy-chair/
doc5 http://www.ikea.com/gb/en/products/sofas-armchairs/armchairs/nolmyra-↘
    easy-chair-birch-veneer-grey-art-10233532/
```

$Cluster_3$

```
doc6 https://headgum.com/the-easy-chair
```

Figure 3.6: Illustration of the clustering step: we present three clusters built from the corpus presented in Fig. 3.5 bottom

A dictionary. This solution is based on a dictionary (noted $\mathcal{D} = \{w_1, .., w_n\}$) which includes the words from the corpus. Naively, we could use all the words that appear at least once in the whole corpus. This would result in a very large dictionary which could impact the execution of the k-means algorithm. In IMAD, we only consider words that appear at least once in all document titles and descriptions. The HTML content, which is the largest part of a document is ignored. This solution is acceptable because the relevant words of a web page are either in its title or its description.

Vectorization. The basic idea is to transform each document doc_j ($1 \leq j \leq m$, m is the number of documents in the corpus) into a n-sized vector (noted vec_doc_j), n be the size of the dictionary. The ith coefficient of vec_doc_j is also called the coefficient of w_i in doc_j. It is noted $Coef_{i,j}$ and it evaluates the importance of the word for characterizing the document.

$Coef_{i,j}$ is computed using the TF-IDF (Term Frequency-Inverse Document Frequency) standard, as follows:

$$Coef_{i,j} = tf_{i,j} \times idf_i \tag{3.1}$$

where $tf_{i,j}$ is the occurrence frequency of w_i in doc_j, and $idf_i = log(\frac{m}{m_i})$, with m_i be the number of documents containing at least once w_i. Roughly, the higher the occurrence frequency of the word in the document, the higher its coefficient. However, the word is penalized in regard to other words if it appears in most of the documents, as it would not be relevant to characterize a specific document.

K setting. K-means requires the number of clusters (k) as an input parameter. [28] proposes a way to calculate k in the context of corpus clustering. It computes k as follows:

$$k = \lceil \frac{m \times n}{t} \rceil \tag{3.2}$$

Where t is the number of non-zero coefficients in the TF-IDF matrix (the stack of all vec_doc_j).

K-means initialization. K-means also requires the initial position of the center of the clusters as an input parameter. A wrong initialization could lead to a wrong result. We use k-means++ [38] to handle this issue. K-means++ is able to choose a satisfying value (not necessarily the optimal one) for the initial center.

3.5.7 Irrelevant cluster elimination

The goal of this step is to discard clusters whose topic does not refer to a trusted entity (e.g. the third cluster in our illustrative example). To this end, we implement the following idea. Each cluster's topic is determined and used to query an accessible database of trusted entities, allowing a score to be assigned to the topic, indicating to what extend it is related to a trusted entity. Then, only clusters whose score exceeds a threshold (determined experimentally) are kept. The challenge here is threefold: cluster's topic determination, finding an accessible database of trusted entities, and determination of an accurate scoring function (noted S).

Topic determination. Given a cluster C, we consider its topic being the list of words which summarizes the main idea developed by all its documents. We call these words *important words* (noted C_{IW}). C_{IW} is computed using the Vector Space Modeling of the cluster. This time, however, the TF standard (Term Frequency) is used instead of TF-IDF as previously. The TF standard is suitable here because we are looking for important words.

Accessible database of trusted entities. The subtle way we adopt is to rely on the two biggest semantic databases that exist: Wikipedia-dbPedia [40] and the World Intellectual Property Organization (WIPO) [41]. We implement a set of tools for accessing the latter. In this thesis we focus on Wikipedia-dbPedia for illustration. Let us say a few words about wikipedia-dbPedia, necessary for understanding our solution. Wikipedia is a participatory-controlled semantic database composed of web pages, called *wiki concepts*. DbPedia is a structured version of wikipedia in which each wiki concept is characterized using several criteria. Among these criteria, the category allows to know if a wiki concept refers to a trusted entity or not. For instance, the category values "Company", "Organization" or "Business" refer to a trusted entity. A wiki concept is also associated to an abstract which quickly describes it.

Cluster scoring. Given a cluster C, its important words C_{IW} are used to compute its score, as follows:

$$S(C) = \max_{w_i \in C_{IW}} (S_1(w_i)) \tag{3.3}$$

where $S_1(w_i)$ is the score of w_i, computed in this way:

$$S_1(w_i) = \max_{wc_k \in WC_i} (S_2(wc_k)) \tag{3.4}$$

where WC_i is the set of wiki concepts related to w_i. WC_i is obtained by

enforcing a Google search only on wikipedia pages ("w_i site:en.wikipedia.org"). The first 10 resulting items are used for extracting WC_i's elements.

Concerning $S_2(wc_k)$, it depends on both wc_k itself and the studied cluster C because two constraints should be respected: (i) If wc_k refers to a trusted entity (whatever it is), $S_2(wc_k)$ has to be high; (ii) If wc_k is unrelated to C, $S_2(wc_k)$ has to be low. This second constraint allows to minimize the false positive rate. For instance, if the topic of a cluster deals with "Orange" (the fruit), the wiki concept "Orange SA" (which is an enterprise) must have a low score. In summary, we compute $S_2(wc_k)$ as follows:

$$S_2(wc_k) = \delta(wc_k) + \sigma(wc_k) \tag{3.5}$$

with $\delta(wc_k)$ be the trusted entity closeness coefficient and $\sigma(wc_k)$ the cluster closeness coefficient. To compute $\delta(wc_k)$, we build a textbook (called the Rescued Category List, RCL) consisting of dbPedia category words which characterize a trusted entity ("Company", "Organization", "Business", etc.) so that: $\delta(wc_k) = 1$ if the category of wc_k appears in the textbook; $\delta(wc_k) = 0$ otherwise.

Note that in our textbook, only categories referring to companies were considered. We focussed on companies since they are the main targets of attackers. However the textbook can be easily enriched with the lexical fields of many other types of organization such as universities and governments agencies.

About $\sigma(wc_k)$, we use the cosine similarity [80], which allows to estimate the distance between two texts. In our case, these texts are: the abstract of the wiki concept (noted abs_wc_k), compared with each document in the cluster. Therefore,

$$\sigma(wc_k) = \frac{\sum_{i=1}^{m_C} cosine(abs\_wc_k, doc_i)}{m_C}$$

.

The application of this step to the illustrative example is as follows. Fig. 3.7 presents the important words of the illustrative clusters. We can see that only the first two clusters are related to trusted entities: EasyChair (the conference management system) and Ikea (furnishing trader).

3.5.8 Irrelevant document elimination

At this stage, all remaining clusters are merged in order to form a unique cluster. The purpose of this step is to focus only on the documents which

$Cluster_1$'s important words

```
easychair , conference , subscribe , sign , management
```

$Cluster_2$'s important words

```
ikea , nolmyra , chair , birch , veneer
```

$Cluster_3$'s important words

```
chair , easy , media , cardiff , supply
```

Figure 3.7: The important words of the clusters presented in Fig. 3.6

directly belong to the trusted entity behind $appName$. To this end, we discard all documents whose domain name is not phonetically close to at least one word of $appName$. For instance, the document *doc2* in the illustrative example does not belong to EasyChair, thus it should be discarded. We choose the domain name because most of the time the company name is used as the basis for building its domain name. We combine two methods to achieve this step: a metaphone algorithm [21] (it assigns to a given string a key indicating its pronunciation) and a string distance algorithm [97] (it compares two strings). Our idea is to first compute the metaphone key of the domain name (noted MK_{dm}) and each $appName$'s word w_i (noted MK_{w_i}). Thereby, we compute the string distance between MK_{dm} and each MK_{w_i} (noted sd_i). Further, it is normalized so that $sd_i = 0$ means the two keys match perfectly. Therefore, if $\exists$ i so that sd_i is smaller than a configured threshold (we experimentally found that 0.3 is a good value, see the evaluation section), the domain name's document is kept. Fig. 3.8 presents the retained documents in the case of the illustrative example.

If the previous step provides no trusted entity, the discarded documents are given a second chance. Indeed, it is possible that an enterprise sells several products whose names are phonetically different from the name of the enterprise (thus its domain name). For instance, the URL of the official website of the famous video game *"Diablo 3"* is *"http://eu.battle.net/d3"*. These cases are rare but exists. They include a category of enterprises that we call *catalogue enterprises* (they offer a catalogue of products). The purpose of the second chance step is to recover them. To this end, we exploit again dbPedia's wiki concept categories (see Section 3.5.7) as follows. We build a textbook (called the Rescued Category List, RCL) consisting of dbPedia categories ("DRM_for_Windows", "DRM_for_OS_X", etc.) which characterize a catalogue entity. Therefore, all domain names whose wiki concept category is part of the RCL are kept. The evaluation results show that this strategy is fairly effective.

```
doc1 http://easychair.org/
```

Figure 3.8: The remaining documents after eliminating those which are not related to EasyChair

3.5.9 Trusted entity's name and contact extraction

At this stage, the remaining documents (if exist) belong to the trusted entity which is behind *appName*. The objective of this step is to determine the contacts of this entity. To this end, we exploit a technique similar to the one described by Google [50]. In fact, websites are usually very clear about who created the content. There are many reasons for this: copyrighted material protection, businesses want users to know who they are, etc. Therefore, most websites have a contact page ("contact us", "about us" or just "about"), copyright information, or include HTML metadata which provide contact information. In nearly 100% of the cases, the company email address is successfully obtained (see the evaluation section). Then, a validation email is sent to the trusted entity and a counter-down is armed. If no response email is received, we suppose the app is illegitimate. The response email should imperatively respect a given format so that it can be parsed by our framework. Notice that if the app is legitimate, the trusted entity will be waiting for the email and will provide a response. For big companies which are subject to several faking, thus will receive a lot of emails, one may think that they will be lost in a myriad of emails. This issue can be easily handled with an email filter. We provide IMAD with such a component which can be easily integrated with popular mail readers.

To circumvent such a validation scheme, an attacker would have to create a fake web site and to exploit search engine optimization to augment the visibility of the web site, so that it becomes more visible than to web site of the company it attacks. Such an attack would be so visible that it would be easily observed and detected by the attacked company (its image is being stolen) which could take counter-measures. Generally, attackers prefer not to behave this way and to remain hidden.

3.5.10 Image Search (based on the logo)

IMAD also leverages the logo of the app (noted *appLogo*) for determining its trusted entity. To this end, we rely on two observations. (1) A logo is strongly linked with a unique app or trusted entity. It is precisely its main purpose. For instance,  uniquely refers to Facebook, the famous social network. (2) The

logo of an app is strongly linked with the trusted entity which possesses it. For instance,  is the logo of both the Facebook corporation and its social network app. We again use Google Custom Search Engine, more precisely its reverse image search system, for achieving this task. By performing a Google search with *appLogo*, we are almost sure to find the trusted entity among the first items. An item here can be of two types: images which are similar to the logo and websites which include the logo. We only consider the second type. Now, the main question is: how to select from the resulting websites the real owner of *appLogo*? To answer this question we implement a six-step algorithm.
Step 1: Construct the corpus, in the same way as presented in Section 3.5.5.
Step 2: Determine the topic of each document doc_i of the corpus, as presented in Section 3.5.6 (Topic determination).
Step 3: Eliminate all documents whose topic is not referring to a trusted entity, as presented in Section 3.5.7 (Cluster scoring).
Step 4: Let us note E_i the owner of doc_i. Search "the E_i logo" on the web. According to the Google ranking system, it is very likely that the logo of E_i is among the first x returned images. We empirically determined that $x = 30$ is fairly sufficient.
Step 5: Compare *appLogo* with each obtained image, using [34]. The latter takes into account several image modifications (rotation, scaling, etc.). The algorithm assigns a rank r_i to every matched image *appLogo*.
Step 6: Choose E_i, so that r_i is the smallest rank. Therefore, E_i is the trusted entity which owns *appLogo*. Its document doc_i is used for extracting its contact, as presented in Section 3.5.9.

3.6 Evaluations

This section presents the evaluation results. We evaluated IMAD from the following perspectives:

- the accuracy: is IMAD able to accurately detect the legitimate entity behind an app's GI? We call accuracy, the ratio between the number of good results provided by IMAD and the number of applications tested during an experiment. Generally speaking, we have a good result if ever: Either the application to be tested imitates a trusted entity and this entity has been discovered by IMAD, or it does not imitate any known trusted entity and IMAD does not detect any entity.
- the scalability: how much resources IMAD consumes?
- the cost: how much money is needed to exploit IMAD?

3.6.1 Experimental environment

The experiments have been realized on two commodity machines, each composed of 4 CPUs (Intel Core i5-3337U, 1.80 GHz), 4 GB memory, an Intel Corporation 3rd Gen Core processor Graphics Controller, and a Gigabit Ethernet card RTL8111/8168/8411 PCI Express. One machine hosts IMAD while the other machine runs a mail server and our mail filter, emulating what should happen in an enterprise. We built a data set playing the role of apps which are in the publication process. This data set is composed of 5,000 apps, organized as follows:

- D_1: includes safe apps gathered from Androzoo [16] and Contagio [36], two popular data sets. These apps are collected from several sources, including Google Play.
- D_2: includes illegitimate apps from Androzoo and Contagio. The latter analyse all apps they collected using different AntiVirus products. Every app in D_2 has its safe version in D_1[8].
- D_3: includes apps we developed for the purpose of this chapter (e.g. Orange Bank) in order to cover all company types (see bellow)[9].

These apps have been selected so that all company types are represented, according to the following criteria:

- the size: big enterprises (BE), and small and medium-sized enterprises (SME).
- the location: developed countries (DC), and emerging countries (EC).
- the activity: catalogue enterprises which are generally gaming companies (GC), and non catalogue enterprises (NGC).
- the name: polysemous (PN) and non polysemous (NPN).

For instance, "Bank of America" is a big enterprise (BE) located in a developed country (DC); it is not a gaming company (NGC) and its name contains polysemous words (NPN). It is said to be of type BE_DC_NGC_PN. Types are equally represented in the data set.

[8]It hasn't always been the case, but that shouldn't change the results, since static analysis systems are mainly based on checking if analysed app signatures have been reported as malware.

[9]As we were unable to develop many apps, we placed only modified names and images of tested companies as IMAD inputs.

3.6.2 Accuracy

The accuracy of our system depends on the accuracy of each step it is composed of, namely: (S_1) GI construction, (S_2) trusted entity search, and (S_3) contact extraction. (S_2), as well as (S_3), has three versions in respect with the three elements which compose the GI (*appName v1*, *appName v2*, and *appLogo*). Therefore, (S_{ij}) corresponds to the j^{th} version of (S_i). The evaluation protocol we put in place is as follows. We manually inspected each app in order to report the expected output of each step. Thus, the actual outputs obtained during the execution of IMAD are compared with the expected values. The accuracy of (S_i) is only accounted if (S_{i-1}) was accurate. We also evaluated the accuracy of the entire system. The latter is accurate when the output of at least one path (among $S_1 \rightarrow S_{21} \rightarrow S_{31}$, $S_1 \rightarrow S_{22} \rightarrow S_{32}$, and $S_1 \rightarrow S_{23} \rightarrow S_{33}$) corresponds to what we manually found[10].

The evaluation includes two experiment types, differing from each other by the considered data set:

- The first experiment type allows to evaluate how IMAD behaves facing safe apps (GI elements have been built without having a malicious idea in mind). We relied on D_1 and D_3.
- The second experiment type allows to evaluate how IMAD behaves facing illegitimate apps. We relied on D_2.

The first experiment type. Fig. 3.9 presents the evaluation results, interpreted as follows. (1) The 100% accuracy rate observed in S_1 and S_3 validates the methodologies we use for extracting GI elements from the apk[11] and contacts from web pages. (2) S_2 is the most critical step. (3) Both S_{21} (based on *appName v1*) and S_{23} (based on *appLogo*) succeed to identify big companies. This is explained by the fact that such companies are well indexed by Google, both their name and logo are popular. (4) The accuracy of S_{23} extends to other company types. This is explained by the fact that the logo is generally built in such a way as to return to a unique company. (5) S_{21} sometimes fails to identify companies which belong to the SME category because they are not so famous as big companies. However, we observed that EC-SME are well identified in comparison with DC-SME. Indeed, it is more difficult to identify

[10]We send a validation email to the 3 identified trusted entities and consider the app submission is validated if one of these 3 contacts responds. Therefore, we consider that the approach is accurate if one of the 3 used methods is successful.

[11]Finally, extraction of salient objects was only done on a sample of logos and not on the graphical interfaces, it was enough to do so.

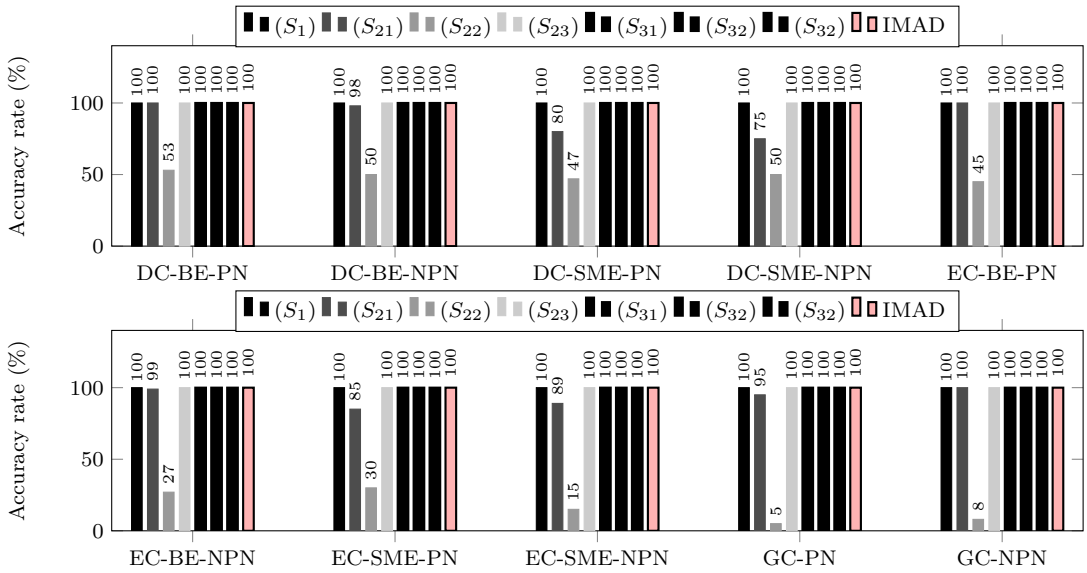


Figure 3.9: First experiment type: evaluation of each IMAD's step with safe apps

a small company in the crowded market of a developed country. This is not the case for an EC-SME which has fewer competitors, thus a better Google index ranking. (6) We also observed that the accuracy of S_{21} is low on apps which belong to countries using a non-Latin alphabet (e.g. Arabic, Mandarin). This situation does not concern big companies of these countries because they most of the time provide an English version of their web site. To validate this observation, we integrate to IMAD a Mandarin string distance algorithm [64]. Then we repeated the previous experiments. We observed the improvement of S_{21} which minimal accuracy rate jumps from about 75% to 93% [12]. The integration of other alphabets to IMAD would nullify the remaining error rate. This is subject of future work. (7) The accuracy of S_{22} (based on *appName v2*) is low because the text extracted from the logo is sometimes completely different from the company name. (8) Considering the fact that at least one IMAD's path is always accurate (the *appLogo* path), IMAD is accurate too.

The second experiment type. We also evaluated IMAD with illegitimate apps. Fig. 3.10 presents the evaluation results. We only focus on the second step. The following observations can be made from these results. (1) S_{21} leads to the lowest accuracy rate, near zero. This is explained by the fact that attackers generally use bizarre names [44], knowing that the right name is overlaid onto the logo. (2) This is why S_{22} and S_{23} provide better accuracy

[12]This is an estimation based on the result obtained after a manual test according to the IMAD methodology on company names in foreign languages.

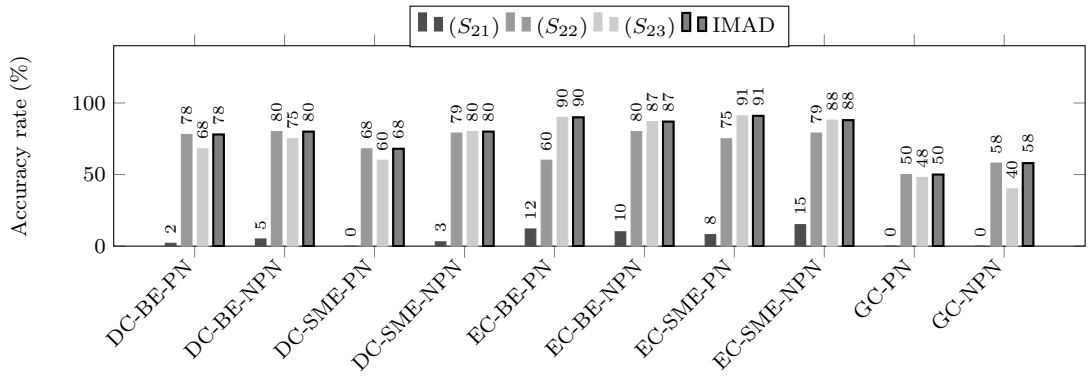


Figure 3.10: Second experiment type: evaluation with illegitimate apps from D_2

rates (more than 80%). (3) Therefore, the average accuracy rate of the entire system is about 80%, which is quite high. This is not so high as with safe apps because some attackers use sophisticated mechanisms for building the GI. For instance, it can be built at runtime by downloading a remote bitmap, making offline solutions inefficient. To handle such cases, we improved IMAD as follows.

Optimization: screen-shots analysis. We improved the GI construction step by extracting salient objects (those on which the human visual system pays more attention) from the first screen of the app. To do so, IMAD runs the app in an emulator and captures the first screen-shot, which has been proven to be enough for identifying an app [90, 63]. Having the screen-shot, we use the algorithm proposed by [34] to extract salient objects. Since not all salient objects are important (e.g. object which represents a text field), we discard all objects representing components which are commonly used in forms (textfield, list, checkbox, etc.). The remaining objects are used for performing web searches. The evaluation results of this optimization reveal a negligible improvement, less than 3%[13]. This is due to the low accuracy of the salient object extraction algorithm[14]. Notice that salient object extraction is a recent and hot topic in the multimedia domain. The evaluation results presented in the next sections rely on IMAD without this optimization.

Comparison with existing solutions

We compared IMAD with two reference illegitimate app detection systems which cover all the existing approaches (Section 3.4.3): source code analysis,

[13]Estimation obtained after a quick implementation and application of the salient object extraction algorithm on some apps in the dataset.

[14]The algorithm we used is the most recent one at the time of writing the paper retated to this chapter.

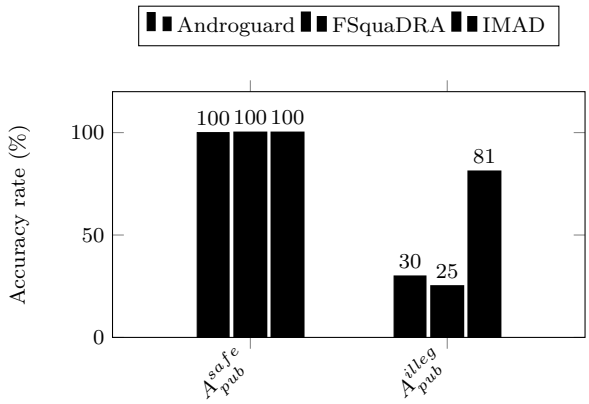


Figure 3.11: Comparison of IMAD with Androguard and FSquaDRA, two reference illegitimate app detection systems

image analysis, and app name analysis. The former is provided by Androguard [17] while the two others are provided by FsQuadra [99]. Recall that the basic idea behind existing solutions consists in comparing the submitted app with those which already exist within the MMP. To compare IMAD with these solutions, we adopted the following protocol. We used half of D_1 as the initial content (yet published apps, noted A^{safe}_{pubed}) of the MMP. The set of apps playing the role of submitted apps consists of the other half of D_1 (noted A^{safe}_{pub}) on the one hand and all apps in D_2 (noted A^{illeg}_{pub}) on the other hand. Androguard and FsQuadra are accurate each time they are able to detect that A^{safe}_{pub} does not mimic an existing app while A^{illeg}_{pub} does. Concerning IMAD, it is accurate when it is able to detect the trusted entity behind the submitted app. Fig. 3.11 presents the evaluation results. We can see that all systems work perfectly on A^{safe}_{pub} apps. Concerning A^{illeg}_{pub} apps, Androguard and FSquaDRA provide poor results. This is explained by the fact that not all A^{illeg}_{pub} mimic apps which are yet in the MMP. IMAD does not suffer from this limitation since its search space is the web. The reader should refer to the previous section in order to have more explanations about the reported accuracy rate (81%).

For legitimate apps, we have 100% accuracy, which means that we are always able to identify the correct entity to contact in order to validate the submission. If this would not be the case, the submitter could contact the MMP for its submission to be handled manually (this would happen only for apps that have no existence at all on the net, which is very rare)[15]. For illegitimate apps, the 20% error rate means that (1) for 80% of these apps, we contacted the (attacked) trusted entity which is therefore informed about the attack and

[15]IMAD inputs for this evaluation are the same as those used for its detailed evaluation in the previous section

Purpose	Value
L_1: Selection of documents which contain exploitable information	20
L_2: Selection of clusters whose topic relates to a trusted entity	1
L_3: Selection of important words	10
L_4: Selection of domain names which are phonetically close to *appName*	0.3

Table 3.2: List of the most important parameters used by IMAD

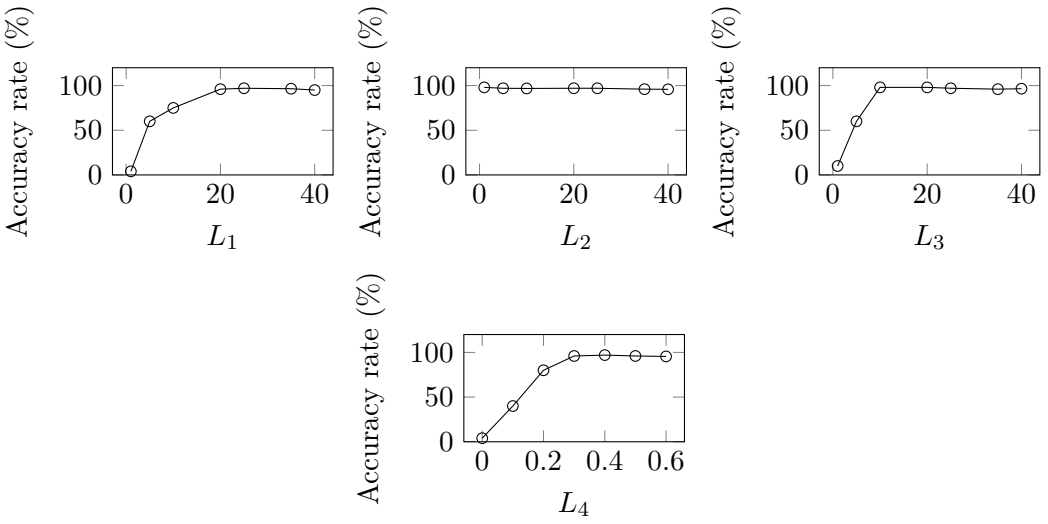


Figure 3.12: Configuration parameters: estimation of the best value

will not validate the submission, and (2) for 20% of the apps, another (wrong) entity is contacted and will not answer or will respond a rejection. The only threat is if an attacker is able to make noise on the web so that it will be more referenced than the trusted entity he attacks. This is a much difficult task for the attacker, as companies which make business with apps always have a communication strategy on the web.

IMAD configuration parameters

Table 3.2 summarizes the list of the most important parameters (noted L_i) that IMAD uses. The last column presents the best value we experimentally found, as follows. For each parameter we performed several experiments while increasing its value. The best value is the one from which the accuracy rate of the system does not improve. These experiments were first performed on a small data set and then validated on the entire data set. The small dataset is disjoint

from the entire dataset which was used for validation. Fig. 3.12 presents the evaluation results, which justify the values reported in Table 3.2. For instance, using more than 20 documents (L_1) in the text search phase does not ameliorate the accuracy rate of the system (see the leftmost curve in Fig. 3.12).

Evaluation of IMAD with other search engines

We also evaluated our system with other search engines namely:

- Bing [30] (from Microsoft) for *appName* searches.
- Tineye [81] for *appLogo* searches.

***appName* searches with Bing.** The custom and programmable version of Bing are fairly new. However, the results obtained with this system are very poor (the accuracy rate is almost nil). Several reasons can explain these results. (1) Bing's ranking does not provide relevant documents at the top. (2) Its API does not allow the specification of the search scope. For instance, it is not possible to enforce a dbPedia search for determining trusted entities. Therefore, although the web search returns some results, it is not possible to classify those which are related to a trusted entity. (3) Small and medium-sized enterprises are almost never found. For these reasons, Bing is not mature enough to be used by IMAD.

***appLogo* searches with Tineye.** We have also tested Tineye [81] as the reverse image search system. We have made the following observations. (1) The Tineye search latency is too high (up to 10sec) in comparison with Google (up to 2sec). (2) Tineye does not implement a ranking system. Therefore, the entity popularity is not taken into account. In order words, the search of a popular entity's logo (such as Facebook) will not necessarily return Facebook's website at the top of the list. However, this system represents an acceptable alternative for Google.

3.6.3 Complexity

This section presents an analysis of the complexity of our approach. It is divided into two parts: local processing and global latency.

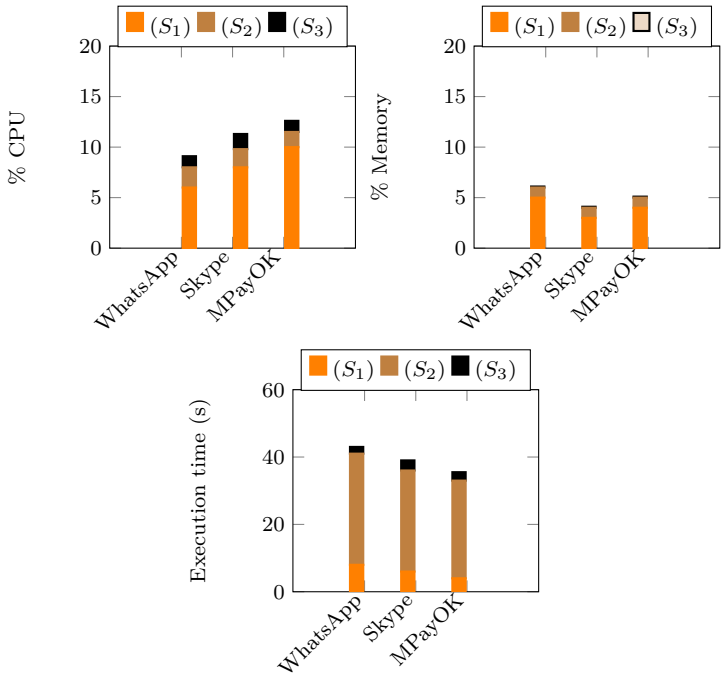


Figure 3.13: Evaluation of the amount of resources consumed by each IMAD's step

Local processing

We consider here the computations on data obtained from remote sources (google and DBpedia). We present separately the complexities of the text search and image search algorithms.

The text search algorithm mainly depends on the number m of documents obtained during the first google engine search. It includes the processing of the collected web pages to build a corpus (Section 3.5.5) whose complexity is O (m), clustering (Section 3.5.6) based on Kmeans with a complexity tending towards $O\ (m^{\ (k\ +\ 1)})$, with k being the number of obtained topics which is generally not more than 5. This step is followed by the elimination of unnecessary clusters (Section 3.5.7) with a complexity O (mnwTN) + 0 (mnNwAM) with n being the number of words in the dictionary, w the average number of wiki concepts relating to an important word, T the number of words in the RCL, N the number of important words, A the number of words in the abstract of a wikiconcept and M the average number of words per document obtained during the first search. The step of eliminating useless documents (Section 3.5.8) has a complexity of O (wT). Finally that of the extraction of the contact (Section 3.5.9) is done in O (m). All the parameters except m being upper bound, the overall complexity

of text search is $O\ (m\ ^{(k\ +\ 1)})$. Finally, let us note that experimentally we have determined that a value of m > 20, does not improve the accuracy of the results.

The image search algorithm consists in the construction of the corpus with a complexity O (m) and the clustering and the determination of the topic whose complexity is $O\ (m\ ^{(k\ +\ 1)})\ +\ O\ (mnwTN)\ +\ 0\ (mnNwAM)$. The step of searching for the logo in the selected documents has a complexity of O (m) and that of comparison of the logos O (xmI) with x the number of logos retained for each document and I the complexity of the logo comparison algorithm. The sorting of the selected logos is done in O (m log(m)). All this for a complexity tending towards $O\ (m\ ^{(k\ +\ 1)})$ for the same reasons as the complexity of text search.

The text search and image search evaluations on several applications have shown that scalability can be taken into account as depicted in Section 3.6.4.

Global latency

The overall latency is dominated by the time taken by each search on Google search engine and on DBpedia. All monetary costs of these searches are evaluated in Section 3.6.5.

3.6.4 Scalability

We evaluated the scalability of our system as follows. First, we evaluated the amount of resources consumed by each step. To this end, we considered three representative apps, namely: WhatsApp, Skype, and MPayOK (mobile money). Then we evaluated IMAD facing parallel checks, up to 10 apps at the same time (Google Play receives about one app every minute [72], thus checking 10 apps at the same time is enough). Fig. 3.13 and Fig. 3.14 present the evaluation results of the two experiment types respectively, interpreted as follows:

- (S_1) consumes a non negligible amount of CPU (this is not the case for other steps), see Fig. 3.13 leftmost. Its memory consumption level depends on the size of the checked apk (10MB-75MB in our experiments).
- When performing several checks at the same time, the processor is the most critical resource since it saturates before the main memory, see Fig. 3.14 left.

- (S_2) is the step which takes the most time, see the last picture in Fig. 3.13. This is because several web searches are performed during this step.
- the average time needed to check an apk is about 38s.

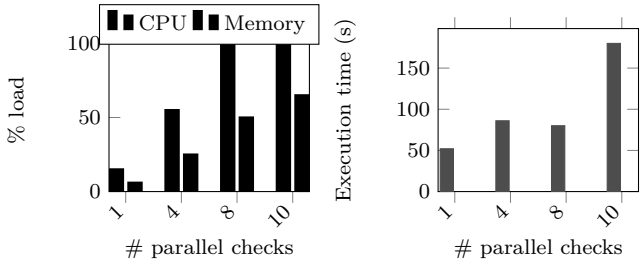


Figure 3.14: Evaluation of IMAD facing parallel app checking: (left) resource consumption and (right) execution time

These results show that the exploitation of IMAD does not require a particular hardware specification, even a commodity desktop can do the job.

3.6.5 Cost evaluation

This section evaluates both the cost of deploying IMAD as an independent service (noted IMADaaS) on a commercial cloud and using Google search engine (which is not free).

Public cloud utilization cost. We evaluated the cost of hosting IMADaaS on a public cloud, Microsoft Azure [66] in our experiments. Our experimental machine (presented in Section 3.6.1) is comparable to the Azure *Standard_A3* virtual machine (VM) type. According to [72, 70], Google Play as well as Apple store receives about 2,000 apps every day (about one per minute). Therefore, a single *Standard_A3* instance would be able to check within one minute the apks coming from up to five MMPs similar to Google Play and App store. This results to a bill of about $1571.28 per year.

Web search engine utilisation cost. The number of searches (noted ns) needed by IMADaaS for checking an apk is given by the following formula:

$$ns = n + k \times m + 2 \times d + n \times d \tag{3.6}$$

where n is the number of items in the corpus, k is the number of clusters, m is the maximum number of important words, d is the number of documents retained for the second chance step. In the case of our experiments, we found the following values: n=20, k=3, m=3, and d=4, resulting to $ns = 117$. We evaluated the cost of using Google custom search engine. The cost of a search in the latter is $0.0075, leading to $0.8775 the cost of checking an apk. Therefore,

the cost for checking all daily apk from Google Play store or Apple store is about $1,755.

3.7 Conclusion

This chapter presented IMAD, a security system capable to identify attackers who deploy malicious mobile applications in the name of well-known public companies which have not deployed their mobile app yet. The IMAD strength lies in the following. The attacker may distort the application label (name, logo) so that it will cheat the detection system (the search engine). However, the label will be distorted in such a way that it is not able to fool the user anymore. We evaluated IMAD with up to 5,000 enterprises, covering all enterprise categories (geographical location, activity, etc.). The evaluation results showed that IMAD can protect both big, medium and small-sized companies with an accuracy rate greater that 80%. We also compared IMAD with two systems (Androguard and FsQuadra). The evaluation results showed that IMAD does better as these systems for classical attacks while they were not able to detect the new attack discussed in this chapter. The evaluation results also showed that our system is able to check all apps deployed within a day on Google Play or Apple App Store at a minimal cost (about $1,755).

This work opens many perspectives and it is worth mentionning some of them. First, the success of this approach depends on the ability to extract the correct email address of the person supposed to validate the application submission. This ability is difficult to experimentally evaluate and if it frequently fails, it would lead to a significant rate of app rejection. Alternatives to this scheme can be investigated. For instance, the submitter could provide an email address and IMAD would verify that this email address complies with the company it has identified, e.g. complies with the DNS domain of that company. One may see a contradiction with footnote 5 in Section 3.1 which argues that we don't trust the submitter. However, if he provides us with an email address and the domain name of this address is that of the trusted entity found by IMAD, the search for the contact can be bypassed and the system can directly send him the validation email. For example, if the given submitter address is submitter0@orange.com, and IMAD finds orange as the trusted entity of the application with the orange.com domain name, we are certain that the developer of the application is an employee of Orange and we will directly send him the validation email. Note that this scheme is secure as long as it is impossible to spoof a domain name. Another perspective is the integration of additional alphabets (we only experimented with Latin and Mandarin in this chapter)

which would improve the accuracy of IMAD. We can also consider using the app's description in our web search for the trusted entity to improve its accuracy. Finally, we are considering is the integration of a cache system which would allow to reduce the costs (in terms of time and money) of web searches.

Chapter 4

Odile: A scalable tracing system for non-rooted and on the shelf Android devices

4.1 Abstract

As Android's popularity continues to grow among consumers and device manufacturers, it is also becoming a prime target for malware authors. Although static app analysis is quite simple to detect malwares and scale very well, it is inefficient when the app is obfuscated or the malicious code is dynamically downloaded at runtime. Runtime analysis of app behavior is thus becoming paramount for users and app market maintainers (e.g., Google Play) to ensure that running apps do not include some malicious payload.

However, in Runtime analysis, dynamic instrumentation, which is a fundamental step to track mobile apps behavior, is very challenging, especially for *off-the-shelf Android devices*.

Most of the time it requires either to root/jailbrake devices, or the use of a modified Android system, preventing its use on any end-user phone. Further, traditional instrumentation techniques do not scale with the number of the intercepted calls as it increases the memory footprint of the instrumented app, ineluctably leading to Out-Of-Memory crash.

We discuss our scalable tracing approach, that we call *delegated instru-*

mentation. It leverages Android's instrumentation module and mainly relies on ART reverse engineering and hacking. We demonstrate the effectiveness of ODILE in tracing various app types (including benign apps and malware) on Samsung Galaxy A7 2017. In particular, we show how much ODILE outperforms FRIDA, the state-of-the-art tool in the domain.

4.2 Introduction

Over the last decade we have witnessed a tremendous popularity of Android smartphones. In particular, Android occupies over 70% of the market share in December 2020 [11]. This popularity goes hand in hand with the fact that it becomes the device of primary importance in the user's daily life, making the smartphone the receptacle of a huge amount of the user's private and sensitive data. Accordingly, it has stirred up envy of hackers who multiply security threats [105, 84] making the Android environment the prime target for malware proliferation to exploit and/or steal data without user consent.

Various techniques have emerged to counter the proliferation of Android malware leveraging either *static*, *dynamic* or *hybrid* analysis. *Static* analysis [56] consists in analyzing the application source code without running it. However, this technique is known for its limitations if applications are obfuscated and/or if their malicious code is downloaded dynamically at runtime. Alternatively, *dynamic* analysis technique [85, 55] comes as a solution to these limitations, and consists in analyzing the actual behavior of the application during its execution. However, due to their inherent high resource consumptions, most dynamic analysis are performed in-lab [86, 8] as opposed to analysis performed on off-the-shelf devices. Accordingly, an in-lab analysis relies on both device emulators and input generation tools (to mimicking user behavior) [15, 86] to automate malware detection, paving the way to anti-analysis techniques. Malware may either recognize the use of an emulator (e.g. based on GPS info, or IMEI number) or detect whether their code is executing in a virtual environment to then hide their malicious payload. Beside, users' behavior simulated by input generator tools can not cover all real user utilization scenarios, leading thus to ignoring several, and potentially malicious, execution paths in evaluated running applications [86]. Finally, *hybrid* analysis techniques rely on static analysis to drive and perform more efficient dynamic analysis by reducing both time and state-space explosion problem [59, 58], but still do not overcome most of the overmentionned shortcomings.

In front of the ever growing proliferation of increasingly sophisticated

malware leveraging on the latest techniques to hide their malicious payload, coupled with vulnerabilities of zero-day threats, dynamic analysis is a fundamental cornerstone to the defense. In this chapter, *we prone that embedding dynamic analysis directly on off-the-shelf devices of end-users* is a very promising alternative to in-lab analysis [71, 79]. It enables to bypass most of the malware anti-analysis techniques while stopping threats such as yet unknown malware variants as soon as possible, i.e as soon as a behavior is detected as suspicious without having identified a particular signature beforehand (i.e. in-labs). However, such a technique is not without shortcomings by its own, and is not applicable to end-users as is. Indeed, existing tracing tools, also named *profiler*, which are at the heart of dynamic analysis, suffers from one key shortcoming: they require either (i) a customized Android system [43, 24, 60, 55], which are not provided with off-the-shelf Android devices, or (ii) a rooted/jailbroken device [71], which is not easily accessible to end users, and can further lead most often to guarantee lost. To confirm this matter of fact, we have manually disassembled and reversed engineered the free version of 7 of the most popular antivirus available from the Google PlayStore (Aegislab, BitDefender, PandaSecurity, Zoner, Drweb, Malwarebyte, and Gdata) following our aim to determine their underlying heuristics to detect malware threats. It appears that they mostly rely on traditional techniques based on signatures. Their advertised dynamic nature comes from their implementation of the *Observer pattern* to observe the external events at the Android system (e.g. permission, file, network observers...) to detect potential suspicious behavior [79]. However, no code patterns related to trace in-memory processes have been found in their disassembled code. It confirms that none of the aforementioned anti-virus leverage dynamic analysis, limiting drastically their ability to detect zero-day attacks or yet unknown variants.

These last years, one promising tools named Frida [7] has gained in popularity due to its ability to instrument dynamically running application on off-the-shelf Android devices without alterations. However, Frida follows a client/sever paradigm, and thus requires the target Android device (that hosts the server part of Frida) to be tethered to a workstation that acts as a client to collect traces from the Android device, and inherently drastically reducing its usability by anti-virus editors. Additionally, Frida does not scale. If it has been designed to successfully hook few functions, it fails to trace more than a few dozen of functions.

To overcomes the above limitations, we introduce the Odile framework that promotes a novel code instrumentation approach, named *delegated in-*

strumentation. ODILE enables to trace Android applications without altering the underlying Android system, as opposed to common *code instrumentation techniques*, which further require a deep application rewriting[37, 71]. Delegated instrumentation requires solely a slight modification to target applications when installed (it is all about one line of code). ODILE has 3 key advantages: it is (i) *lightweight*, (ii) *scalable*, and (iii) *interoperable*. Particularly, it consumes few resources, it can trace thousands of method calls, and it can run irrespectively on most phones (x86 and ARM) from the market (i.e. ODILE does not rely on any proprietary software substrate specific to some smartphone manufacturers to trace seamlessly processes).

In summary, this chapter makes the following contributions:

- We introduce an open source framework ODILE that seamlessly trace Android applications on off-the-shelf Android x86 and ARM devices without any alterations at the underlying Android system. Particularly, ODILE is smoothly intertwined with the underlying Android runtime ART to leverage on its internal features to promote what we call a delegated instrumentation technique that enables to trace any Android application process flawlessly.
- We have thoroughly evaluated the ODILE framework. We have highlighted its effectiveness and scalability compared to the top-notched solutions such as FRIDA. In particular, ODILE does not incur any memory and cpu overheads while it can scale to several thousands of traced method calls.

The rest of the chapter is organized as follows. Section 4.3 presents the necessary background. Section 4.4 presents the motivations. Section 4.5 presents the ODILE and its implementation. Section 4.6 presents the evaluation results. Section 4.7 presents the related work. Section 4.8 concludes the chapter and presents future works.

4.3 Android

This section presents the necessary background to understand our contributions.

Generalities. Android is a mobile-targeted Linux-based operating system (OS) augmented with a set of system services (essentially written in C and C++) which run at the user space level. These system services are continuously evolving over time to accommodate with new functionality requirements. A

significant evolution has been the replacement of the historical Dalvik [4] interpreter. Prior to version 4.4, apps were compiled to a bytecode format called Dalvik EXecutable (`dex`), which were directly interpreted by a Dalvik virtual machine. Since version 4.4, Android RunTime (ART) is used instead. ART works on Of Ahead Time (`oat`) files, composed of the ELF shared object file obtained from a `dex` file using `dextooat` compiler, and the `dex` file. The latter has been kept in Android because the interpretation logic is still required for some purposes such as method instrumentation, the service that ODILE leverages.

App startup. The launcher, which is a system service provided as an app, orchestrates any app startup process. Once the user click on an app icon, the launcher asks the *Activity Manager Service (AMS)* to start the principal app's activity. To this end, the AMS forks the pre-warmed process `Zygote` to quickly setup an execution environment for the app. The app's `oat` file is loaded into the forked process's address space. Besides, memory addresses of the objects which allow ART manipulation (e.g., `ArtMethod`, `ArtClass`) from the app's address space are loaded. Finally, the principal app's activity is shown to the user.

Method call and execution. In Android, the execution of any method involves three central components: the system service which wants to invoke the actual method (e.g., ASM), the `ClassLoader` and ART (which was already loaded within the app's address space). The `ClassLoader` is the workflow orchestrator. For illustration, let us consider the execution of `onCreate()` from the principal activity, initiated by the AMS. The `ClassLoader` first looks for the class which implements `onCreate()`. Then it instructs ART to locate `onCreate()` from the app's `oat` file and to invoke it. The calling path is as follows:

```
art::Invoke_***_method(MainActivity, oncreate)->[ART-
    representation-of-onCreate]::Invoke()->...(some method
    arguments preparation, other stuff)...->
    art_quick_invoke_stub...->Oat_quick_method_code()}
```

where `Oat_quick_method_code()` is the field member of `ART-representation-of-onCreate` C++ object, the latter holding the memory address of the invoked method; `art_quick_invoke_stub` makes the bridge between ART's functions and app's `aot` memory address range.

The instrumentation module. ART embeds an instrumentation module which is used at development time. It is activated and instructed by the

debugger (*dgb*). This module allows to intercept and to trace method calls during the execution of an app. It takes as input a set of functions, called *listeners*, that will be executed each time a target function is invoked. To this end, the instrumentation module can use two approaches selected by the debugger. In the first approach, it replaces `Oat_quick_method_code()` (which is the entry point of any method call as seen above) by `art_quick_instrumentation_entry()`. The latter is a kind of bridge between the execution of the app and the instrumentation module. This way, any call to a method that needs to be traced first invokes listener functions. Without loosing the details, notice that this approach implements a stack logic to deal with internal calls performed by the traced methods. The second approach relies on ART's interpreter. The latter is activated using two ART's functions: `enableDeoptimization()` and `deoptimizeEverything()`. This second approach consists in replacing the entry point of all methods that need to be traced by the bridge function `art_quick_to_interpreter_bridge()`. Thus, once inside the interpretation loop of an app's method, if the `doCall` instruction is encountered (which means a method call), listener functions are invoked before the target method. Our tracing system ODILE leverages this second approach.

4.4 Frida limitations

To the best of our knowledge, FRIDA [7] is the only existing dynamic binary instrumentation tool for non-rooted off-the-shelf Android devices. To avoid phone rooting (to install a custom kernel), FRIDA leverages the fact that ART is part of the address space of any Android running app. This leads ART *hackable* from the original app code base. Hacking a function consists of dynamically surrounding or replacing the function by a piece of code which is generally named *hook*. We categorize this hacking approach as *code instrumentation* in the sense that the hook is *inlined* at the target function location.

FRIDA uses a client-server model in which the server component runs on the smartphone while the client runs on a machine. The two equipments are linked with a usb cable. Given a target apk that needs to be instrumented, FRIDA is used as follows. First, the user modifies the apk in order to make FRIDA's server being the new app entry point. Then she installs and launches the apk on the smartphone. On startup, FRIDA's server opens a socket for listening instrumentation orders sent by the client side. FRIDA allows to use java script for writing the instrumentation orders. The latter consists of two elements: a set of function signatures that we want to hack, and for each function the corresponding hook. During the instrumentation campaign, the

output generated by the hooks are forwarded to the socket.

Before we decided to build ODILE, we first tried to use FRIDA as a tracing tool for non-rooted off-the-shelf Android devices. This could have been possible by asking FRIDA to hack all functions using the same hook which just prints each function identity (name and actual parameter values). To this end, we brought two main improvements to FRIDA. First, we embedded the client component into the smartphone. Second, we ported FRIDA to ARM, thus covering the vast majority of smartphones. In the rest of the document, the term FRIDA refers to this new version that we built.

However, we faced with a conceptual limitation of FRIDA: *code instrumentation* does not scale in the context of tracing because that latter activity involves thousands of function calls. We empirically observed that FRIDA crashes early, after about 200 function calls. We found that these crashes are due to the classical OOM kill event. Section 4.6 presents more evaluation results.

In addition to the scaling issue, *code instrumentation* is inefficient for tracing functions whose payloads are dynamically downloaded at runtime. In fact, these functions are not known at app launch time, thus they cannot be instrumented. Knowing that several malware use to dynamically download their malicious payload, a *code instrumentation* based tracing tool will miss relevant calls.

The next section presents ODILE, a tracing tool for non-rooted off-the-shelf Android devices that we build based on FRIDA while adopting a different instrumentation method.

4.5 Odile

The goal of ODILE is to trace/monitor directly on the phone the calls performed by a target app. We first present the tracing approach behind ODILE. Then we present the general workflow and architecture of ODILE. Finally we detail the implementation of ODILE, which targets ARM and x86 devices, thus covering almost all existing smartphones.

4.5.1 Main idea

ODILE combines *code instrumentation* (for a few number of actions, see below) with a new approach that we call *delegated instrumentation* to trace a

target function. To this end, ODILE leverages two specificities:

- related to the tracing discipline: a tracing system uses a unique hook for all intercepted calls, contrary to a general purpose code instrumentation tool such as FRIDA.
- related to ART architecture: ART natively includes an instrumentation module which, when activated, can automatically invoke a registered callback function every time an app function is called.

ODILE uses code instrumentation for dynamically hacking ART in order to enable the instrumentation module and to register on the fly a generic hook to the latter. The actual tracing of app's functions is therefore *delegated* to ART's instrumentation module. Although this idea is easy to label, its implementation is tricky.

4.5.2 Architecture

ODILE is intended to be used by behavioral analysis apps (such as anti-virus), but not directly by phone owners. By default, ODILE intercepts all calls performed by the target app. However, it can be configured to monitor only a given set of functions which signatures are provided to ODILE. Fig. 4.1 presents the general architecture and workflow of ODILE. ODILE is composed of four main components namely: the *Hacker* (for ART and its instrumentation module hacking), the *Tracer* (for function call tracing), the *TraceCollector* (for collecting traces) and the *Orchestrator* (which orchestrates the whole workflow). We use the terms ODILE and *Orchestrator* interchangeably. Let us note A_{target} an app that we want to trace using ODILE. ODILE intervenes at two moments during A_{target} lifetime: at installation time and at runtime. ODILE (which runs in background along with the behavioral analysis app which uses it) is informed by the Android system when A_{target} is installed (Fig. 4.1 left side). To this end, ODILE registers a `BroadcastReceiver`[1]. Upon receiving the installation intent, ODILE dynamically integrates (*repackaging* phase) into A_{target} both the *Tracer* and the *Hacker* payloads. (Note that we include the latter ODILE's resources (files that can be used at runtime).) ODILE uses `Soot` [20] as the repackaging tool, although other tools like jadx [9] can also be used.

ODILE integrates *Tracer* into A_{target} as a resource while it does code

[1]Notice that the OS can delete a registered `BroadcastReceiver` when an app runs in background, which is the case of ODILE. To avoid this situation, ODILE relies on an Android `JobIntentService` for `BroadcastReceiver` registration.

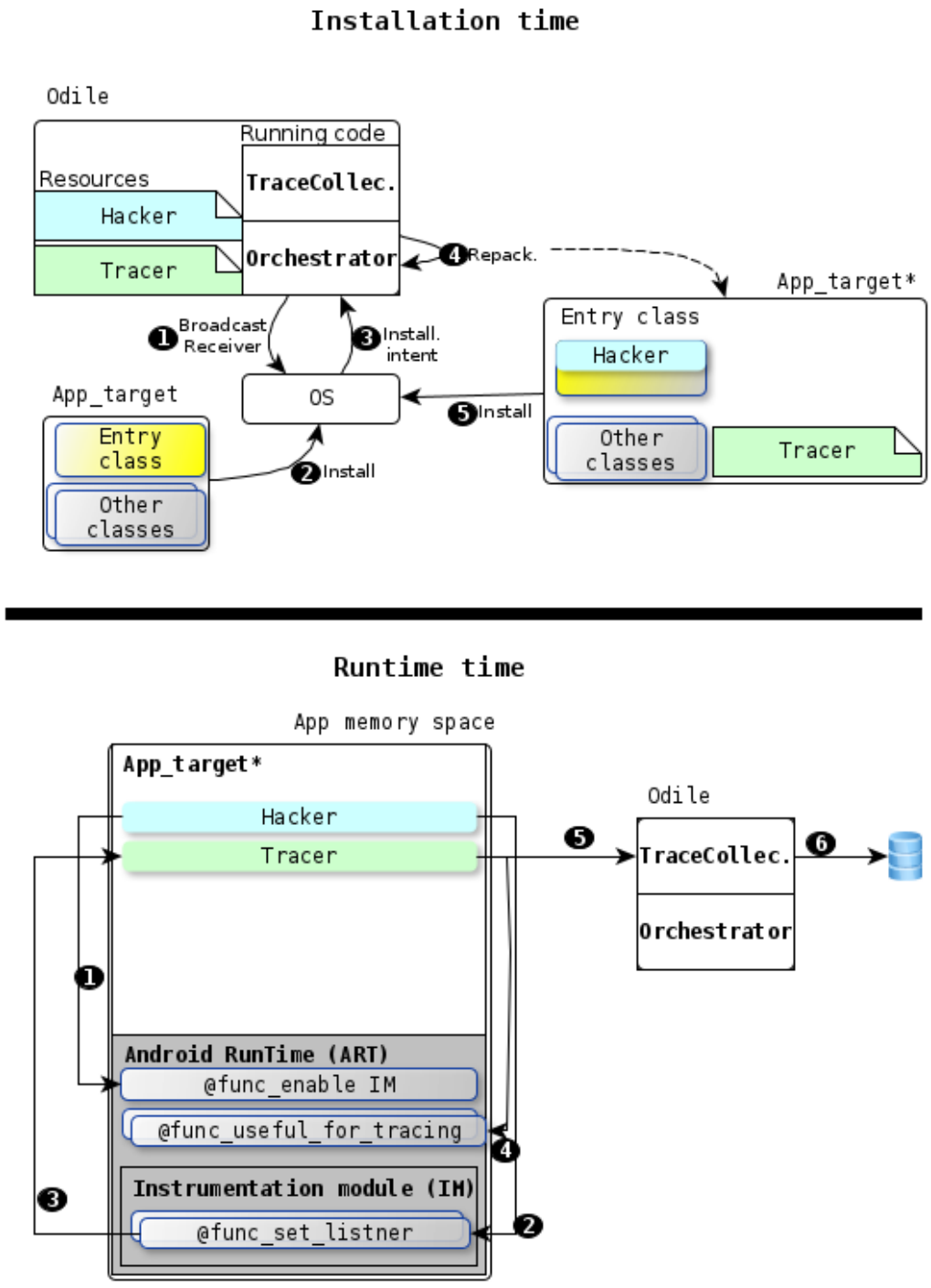


Figure 4.1: Odile general architecture and workflow. Odile is provided as a classical app. To trace a given app (noted A_{target}), Odile intervenes at two moments: (up) app installation time and (bottom) runtime.

injection for the *Hacker* in order to court circuit A_{target} bootstrap phase. The purpose of the *Hacker* is to activate and configure ART's instrumentation module (actions 1 and 2 in Fig. 4.1 right side) at A_{target} boot time. The configuration of the instrumentation module is to register the *Tracer* as the callback function. This way, when invoked latter by the instrumentation module (step 3 in Fig. 4.1 right side), the *Tracer* will compute the relevant informations related to actual A_{target}'s function call at the origin of the invocation. The informations that we consider are: the **function name**, its memory address, its **argument types** and their **actual values**.

Except the memory address of the traced function, which is directly provided to the *Tracer* by ART's instrumentation module, the computation of the other informations are more tricky. We mainly rely on ART reverse engineering and hacking (step 4 in Fig. 4.1 right side), that we use in a global strategy as follows. Given an information (function name, argument types or values) that we want to compute:

(1) *(phase 1)* we explore ART source code in order to obtain a set of public functions which will help computing the desired information (e.g., function name). This phase returns two elements: a set of functions (each represented by a triplet) noted $\mathcal{S} = \bigcup_{i=1}^{n} (f_i, obj(f_i), arg\_list(f_i))$ and a call graph of these functions, noted $\mathcal{G}(f)$, whose execution outputs the desired information. $\mathcal{G}(f)$ can be seen as the strategy that we use to compute the desired information; n is the number of distinct functions in the call graph; f_i $(1 \leq i \leq n)$ are those functions; $obj(f_i)$ is the C++ object which declares f_i, and $arg\_list(f_i)$ is the list of f_i's arguments. We realize this phase at ODILE design time.

(2) *(phase 2)* we designed seven strategies (see *Strategy 1-7* in the remaining sections) to obtain the memory address of each element in $\mathcal{S}$. Indeed, these addresses are mandatory for the execution of $\mathcal{G}$ (see the next step). Therefore, the output of this second phase is noted $\mathcal{A} = \bigcup_{i=1}^{n} (@f_i, @obj(f_i), @arg\_list(f_i))$, where @ means memory address. The strategies that we use can be organized into three categories in respect with their execution moment: at ODILE design time, runtime, or hybrid.

(3) *(phase 3)* The last phase, performed at ODILE runtime by the *Tracer*, is the actual execution of $\mathcal{G}$. This is done by hacking ART (step 4 in Fig. 4.1), see the next section. The output of this phase is the desired information (e.g., function name).

The next sections detail the implementation of ODILE's components, phases

and strategies.

4.5.3 ART's function hacking

Given an ART's function, described by the triplet $(@f, @obj(f_i), @arg\_list(f_i))$, this section describes how the *Hacker* invokes[2] it at runtime. To avoid reinventing the wheel, the *Hacker* relies on FRIDA's hacking stack, summarized in Fig. 4.2. The hacking stack exposes a high level javascript (js) API called `NativeFunction`[3]. The goal of the latter is to configure calling parameters for the low level library `libffi`, which is the actual one that invokes the ART's function that we want to invoke/hack. `NativeFunction` has a dual implementation: a JS version and a C version. The former is a wrapper for upper level components such as the *Hacker* (as shown in Fig. 4.2). It calls the C version which actually implements the core (e.g., type conversions from the JS world to the C world). The binding between the two versions is ensured by a JS engine, which is `duktape` [6] in ODILE. About `libffi` [10] library, it provides a portable, high level programming interface to various calling conventions. Its allows to call any function specified by a call interface description at run time. Given $(@f, @obj(f_i), @arg\_list(f_i))$, `libffi` is able to invoke the corresponding function.

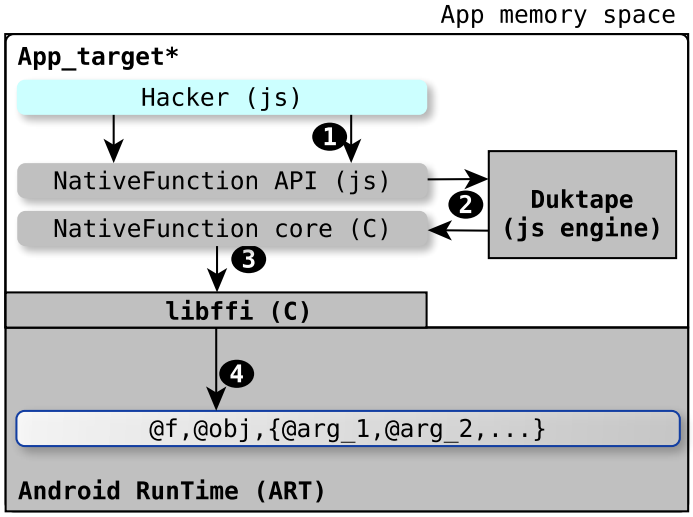


Figure 4.2: Hacking stack

[2]Note that this is a function hacking operation in which the hook is empty.

[3]FRIDA requires the user to provide the hook. To simplify the writing of the latter, FRIDA allows to implement it in JS, which is popular in within the scope of mobile app programing, compared to low level languages such as C and C++.

4.5.4 ART's instrumentation module activation

ART's instrumentation module is activated by the *Hacker*. As presented in Section 4.3, this can be done by invoking `enableDeoptimization` and `deoptimizeEverything`, two methods of the object which represents the instrumentation module (*instrumentation*). The challenge here is how to invoke these two methods from the *Hacker* code base knowing that they belong to ART?.

Making ART's functions accessible from the Hacker. To this end, the *Hacker* first preloads `libART` by calling `dlopen` from *libc*. Notice that `dlopen` just calls `realDlopen`, which is the core implementation. However, `realDlopen` implements a security check which accepts its invocation only when the caller of `dlopen` (the *Hacker* here) is within `libART`'s address range, which is not the case of the *Hacker*. To bypass this limitation, we write a trampoline function which replaces `dlopen` and calls `realDlopen` with a fake caller address which address is within `libART`'s address range. We use the address of `read()` syscall.

(Strategy 1) We call this strategy wrapper-based function call.

enableDeoptimization and deoptimizeEverything memory address determination. Having `libART` accessible, the *Hacker* uses `dlsym` to obtain the desired addresses. `dlsym` takes as input the mangled name of the functions. We manually obtained these mangled names by reverse engineering ART binary code. For example, the mangled name of `enableDeoptimization` is "$\_ZN3art15instrumentation15Instrumen...$". Listing 4.1 illustrates an utilization of `dlsym` to obtain the memory address of `enableDeoptimization`.

Listing 4.1: `enableDeoptimization` address calculation.

```
2 const enableDeoptimization: any = new NativeFunction(
3  dlsym(artlib,"_ZN3art15instrumentation15Instrumen..."),
4   "void",
5      ["pointer"],
6      {
7         exceptions: ExceptionsBehavior.Propagate
8 })
```

(Strategy 2) We call this strategy `dlsym`-based address calculation.

Instrumentation module memory address determination. We implement a two step solution: (1) determine *Runtime*, the object which represents ART; (2) determine *instrumentation*, the field member of *Runtime* which represents the instrumentation module. We implement the first step as follows. We manually identify in ART source code a public function that uses *Runtime*; we choose `RequiresDeoptimization` method, lines 1-6 of listing 4.2. Having such a function, we can identify in its assembly code the portion that yields the address of *Runtime* (lines 7-15 of listing 4.2 for x86 architectures). By reverse engineering, we know that on x86 architectures for example, the address of *Runtime* will be held in $x8$ register. Then we alter the previously identified portion of code so that its execution will assign the address of *Runtime* in a new local variable that we introduce, instead of register $x8$. We call *parasite* the resulting code (lines 16-24 of listing 4.2). Then we integrate the parasite into the *Hacker* code base. This way, at runtime, the *Hacker* will be able to fetch *Runtime* memory address from the local variable mentioned above.

(Strategy 3) We call this strategy emulation-based address computation.

The instrumentation module (*instrumentation*), which is a field member of *Runtime*, can be obtained by just adding an offset to *Runtime*'s memory address. This offset is related to the position of *instrumentation* in the declaration of *Runtime* class. We obtain this offset manually by reverse engineering ART binary code.

(Strategy 4) We call this strategy position-based address computation.

Listing 4.2: Computation strategy of the Runtime object

```
1 ------------------------------------
2 C++ code of RequiresDeoptimization()
3 ------------------------------------
4 bool Dbg::RequiresDeoptimization() {
5   return !Runtime::Current()->GetInstrumentation()->
        IsForcedInterpretOnly();
6 }
7 ------------------------------------
8 asm code of RequiresDeoptimization()
```

```
9  ------------------------------------
10 inst_0 : adrp x8, #0x78459ac000
11 inst_1 : ldr x8,[x8, #0x2e0]
12 inst_2 : ldr x8,[x8]
13 inst_3 : ldrb w8, [x8, #0x2dc]
14 inst_4 : cmp w8, #0
15 inst_5 : ...
16 ------------------------------------
17 The parasite inside the Hacker
18 ------------------------------------
19 let 0x2e0 = 736;
20 const operand = insn_0.operands[1].value;
21 const x8 = Memory.readPointer(operand.add(0x2e0));
22 const x8 = Memory.readPointer(x8);
23 return x8;
24 ------------------------------------
```

Instrumentation module activation and configuration. The activation is done by invoking `enableDeoptimization` followed by `deoptimizeEverything`, two functions of *instrumentation* object. Then the *Hacker* configures the instrumentation module with the list of monitored events and the associated callback functions. In ODILE, we monitor function entrances and we use a unique callback which is the *Tracer*.

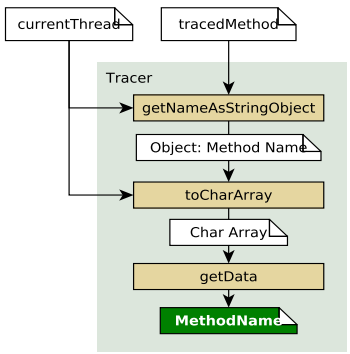


Figure 4.3: Method name computation.

4.5.5 Function call information retrieval

When ART's instrumentation module invokes the *Tracer*, it passes to the latter three parameters including the object which corresponds to the current virtual machine thread (noted $currentVM$), the object which corresponds to the traced method (noted $tracedMethod$), and the object from which

tracedMethod is invoked (noted *caller*). Using these arguments, the *Tracer* computes and outputs in a *human readable format* the following informations: *tracedMethod*'s name, the types of its arguments, and their actual values. The challenge here is that these informations are disseminated everywhere in the call stack, which is not always directly accessible from the *Tracer*. The rest of this section presents the strategies that we use to obtain each information.

tracedMethod's name computation. By reverse engineering ART, we observe that the execution of the call graph presented in Fig. 4.3 can allow obtaining the *tracedMethod*'s name. This call graph takes as input *currentVM* and *tracedMethod*, and it calls the following three ART's internal methods, in this order: `getNameAsStringObject`, `toCharArray`, and `getData`. We obtain the memory address of each of these functions (to invoke them) using *Strategy 2* presented in § 4.5.4.

tracedMethod's argument list address computation. By reverse engineering ART, we observe that *tracedMethod*'s argument list can be found in ART's object *shadow_frame*. We use the following strategy to obtain the latter. We know that *shadow_frame* is an argument of ART's internal function `EnterInterpreterFromEntryPoint`. We also know that the latter is in the call stack of the current executed function. Subsequently, *shadow_frame* can be obtained by exploring the current call stack. On x86, this is quite simple because the call stack is located in the current virtual machine's memory stack.

> *(Strategy 5)* We call this strategy stack-based address computation.

Things are different on ARM. We rely on a secondary instrumentation callback that ODILE registered at the same moment as the *Tracer* (see above). The purpose of this secondary callback (summarized Fig. 4.4) is to track the invocation of `EnterInterpreterFromEntryPoint`. On invocation of `EnterInterpreterFromEntryPoint`, the secondary callback captures the address of *shadow_frame* and saves it in a new ART's global variable that we have dynamically introduced. This variable is latter read by the *Tracer* to obtain *shadow_frame*'s address.

> *(Strategy 6)* We call this strategy callback-based address computation.

As mentioned above, the memory address of the arguments of *tracedMethod* are kept in a *shadow_frame*'s field member in the form of a list. We obtain this list using *Strategy 3-4* presented in § 4.5.4. However, the list contains not

only *tracedMethod*'s arguments, but also other values specific to the virtual machine ABI such as the return address. By reverse engineering ART, we know that the position of the first *tracedMethod*'s argument in the list is given by *code_item* object. The latter is an argument of another ART's function named `ArtInterpreterToInterpreterBridge`. We obtain *code_item* using *Strategy 5 or 6* in respect with the hardware.

tracedMethod's argument types and actual values computation. We need argument types for two reasons. First, they allow ODILE to output a complete (thus unique) signature of the traced method, necessary to identify a function when function overloading exists in the app. Second, the argument type allows to extract the actual value of the argument having its memory address. In fact, the argument type gives the argument value size in RAM. To find the type of each argument, the *Tracer* uses ART's function `getShorty()`. However, this function is private in *Runtime*, thus it cannot be directly invoked from outside that object. In addition, `getShorty()` is an inlined function, meaning that we cannot have any direct reference to it. To overcome these limitations we implement the following strategy. First, we identify in ART binary code a function which inlines `getShorty()`. Such a function is `Dbg::OutputMethodReturnValue` for x86 and `art::QuickGenericJniTrampoline` for ARM. Then we identify the portion of code corresponding to `getShorty()`. Using *Strategy 3-4* presented in § 4.5.4, we build a parasite from that portion of code and we integrate it into the *Tracer*. Therefore, the later execution of the parasite with a *tracedMethod*'s argument address as the input argument generates the actual value of that *tracedMethod*'s argument.

(Strategy 7) We call this strategy inliner-based code detection.

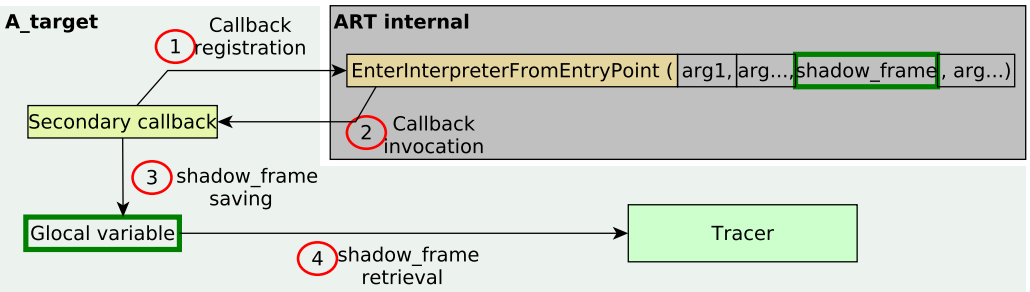


Figure 4.4: *shadow_frame*'s memory address computation on ARM

Trace collection. Recall that unlike the *Hacker* and the *Tracer* which are

embedded into App_{target} by repackaging, the $TraceCollector$ runs inside ODILE app. The informations generated by the $Tracer$ are sent to the $TraceCollector$ throughout a TCP/IP connection. The $TraceCollector$ can save the collected data either in the phone file system (default) or on a remote server.

4.6 Evaluations

This section presents the evaluation results of ODILE. The evaluations answer the following questions:

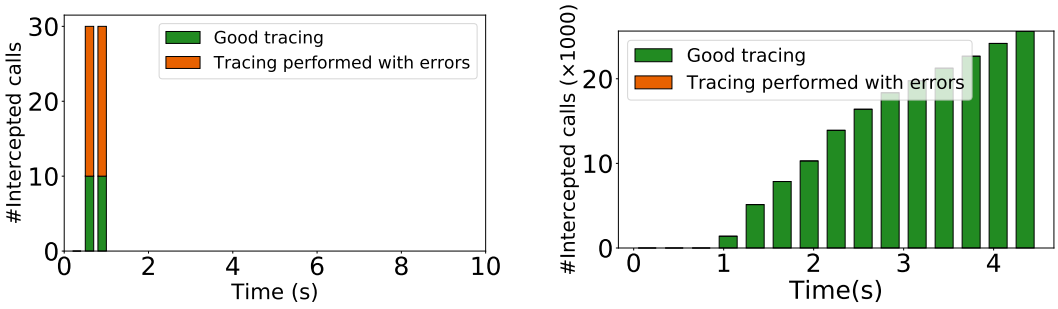
- *Is* ODILE *effective in intercepting app function calls?* We answer this question by using a realistic experimental environment, as follows. First, we carry out experiments on Samsung Galaxy A7 2017, with 8 cores, 4GB memory, 64GB storage, and which runs Android 8.0. Second we use 18 benign apps (7 popular apps from F-Droid [1], and 7 anti-virus: Aegislab, BitDefender, PandaSecurity, Zoner, Drweb, Malwarebyte and Gdata) and 58 malware (from [5]). We chose these apps in each dataset because they are compatible with the repackaging tool (Soot) used by ODILE. Also note that F-Droid [1] is a popular Google Play alternatives which provides app source code. Thus, we can manually insert tracing functions into the app in order to build a baseline for evaluating ODILE effectiveness and completeness.
- *How scalable* ODILE *is?* We answer this question by varying the number of functions that we want to trace.
- *What is the overhead of* ODILE*?* We measure both CPU and memory consumption incurred by ODILE.
- *What is the impact of* ODILE *on user experience?* We measure the impact at app installation time, launch time, and utilization time. This impact corresponds to the additional delay incurred by ODILE.

We compare ODILE with FRIDA when necessary. Each experiment is repeated 5 times and the mean value is considered.

4.6.1 Odile effectiveness

For this experiments we use both benign and malware apps.

Basic evaluation. To validate that ODILE is effectively able to intercept and trace all app calls we use the following methodology. To have a baseline, we manually instrument Textpad [14], an open app from F-Droid. Then we run



(a) Frida (b) Odile

Figure 4.5: Odile effectiveness on DualOps app, compared with Frida.

Textpad and we collect the logs (called *logs-1*) generated by our inserted print functions. On the second hand, we run vanilla Textpad with Odile and we collect the generated logs (noted *logs-2*). We observe that *logs-2* covers all calls in *logs-1*, validating the effectiveness of Odile.

Comparison with Frida. Frida takes as input a set of function signatures that we want to monitor. Therefore, we use Odile in this same way. For this experiment, we use all working apps in our testbed. For each app, we randomly select 1,800 distinct functions and we use them as the input of Frida and Odile. Table 4.1 summarizes the results for a representative set of apps while Fig. 4.5 focuses on DualOps app[4]. Fig. 4.5 shows the evolution of the number of intercepted function calls in both systems during the experiment. We observe two kind of issues with Frida (Fig. 4.5a), contrary to Odile (Fig. 4.5b) which has no issues. The first issue is the fact that Frida returns errors for some intercepted calls. These errors represent up to 50% of the total intercepted calls (see the red part of the bars in Fig. 4.5a) The second issue is the abrupt crash of Frida during the experiment. This occurs almost 1.3 second after the app startup, leading to a very few number of intercepted calls (about 30) when compared with Odile (about 25,000 as shown in Fig. 4.5b). Odile never crashes, thus we voluntary stopped the experiment after 4.7 seconds. We observe similar results with other apps as shown in table 4.1.

Anti-virus dynamic analysis. The objective is to observe if the anti-virus monitors and analyses the behavior of apps which run on the device. Given an

[4]We randomly select DualOps, which is a randsomeware. At startup, it presents to the user an activity that blocks the phone while displaying a message accusing the user of having stored and viewed prohibited content. A payment procedure of the fine also shown.

Table 4.1: ODILE (Od) tracing effectiveness compared with FRIDA (Fd). #calls is the total number of intercepted calls. #capNotTraced is the number of intercepted calls that FRIDA was not able to trace (FRIDA prints an error message in this case).

apps	#calls		#capNotTraced		Crash (yes/no)	
	Fd	Od	Fd	Od	Fd	Od
Anigilyator	64	21k	64	0	yes	no
superLinda	8	5k	7	0	yes	no
Anton	48	22k	48	0	yes	no
DualOps	31	25k	21	0	yes	no
anigilyatorV2	36	18k	23	0	no	no
LockApp	27	19k	27	0	yes	no

app that needs to be installed on the phone, our anti-virus analysis methodology is as follows. We collect anti-virus's execution traces at different moment of the app lifespan: installation time, launch time and utilization time. We present here the results for *Zoner* anti-virus (for illustration) when it is used to scan DualOps. At DualOps installation, the execution traces of the anti-virus show that it performs two main calls: `InstallReceiver->onReceive()` followed by `DexParser->scanAPK()`. By reverse engineering the anti-virus, we know that the purpose of these calls is to append the new installed app into a file for static analysis (see below). We observe no behavior change in the execution of the anti-virus neither at DualOps launch time nor during its utilization. In fact, the anti-virus periodically performs these calls: `ActScanResults->onBound()`, `HashInputStream->read()`, `DbScanner->compareTo()`, `ActScanResults->addResults()`, in this exact order. By reverse engineering these calls, we observe that they realize a static analysis based on DualOps `dex` file. To conclude, the answer to our above question is that the anti-viruses that we analyzed do not perform app dynamic analysis on the phone as evoked in Section 4.2.

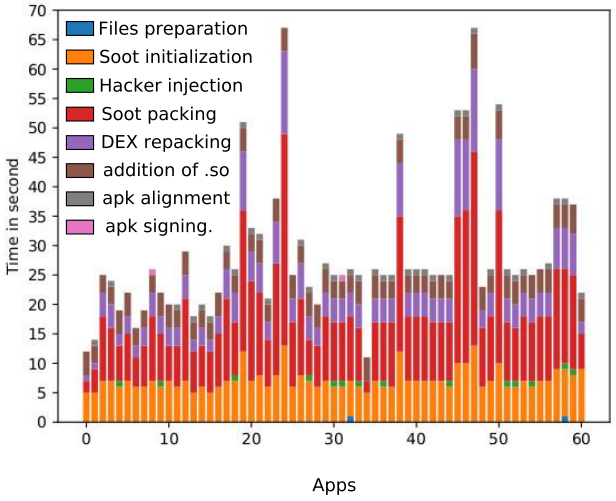
4.6.2 Overhead

We evaluate the impact of ODILE on app installation time and user experience at runtime.

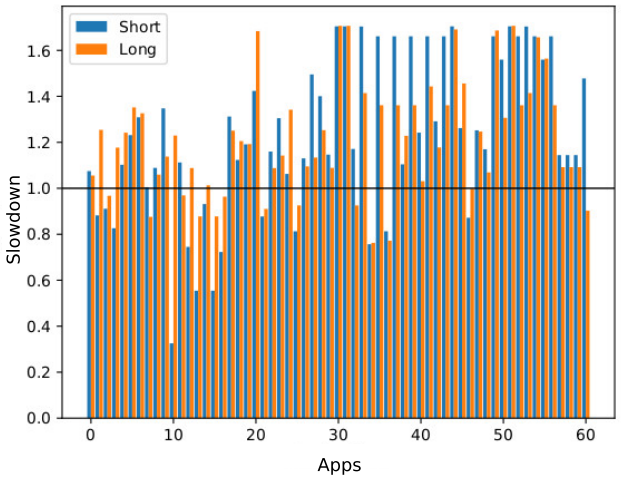
Installation time. The overhead here is mainly the repackaging step imposed by ODILE. Fig. 4.6a presents the breakdown time of the repackaging step. This step is organized into 8 phases. The total repackaging time varies between 10 seconds to 65 seconds, knowing that the installation time without ODILE is around 25 seconds. We can see that the phase 4 (*Soot* packing) is the longest one, followed by phase 2 (Soot Initialization) and phase 5 (Dex

repackaging). Other phases are relatively shorter. We note that the size of the apk does not impact the repackaging time. Indeed, Fig. 4.6a presents apps in the ascending order of their apk size, which varies from 56KB to 4.1MB.

Runtime. Fig. 4.6b presents the impact of ODILE on the launch time. The launch time is the time it takes to display the first app's activity. We compute the impact of ODILE as the difference of the launch time with and without



(a)



(b)

Figure 4.6: (a) Breakdown time of ODILE's repackaging step and (b) Slowdown to display the first app's activity.

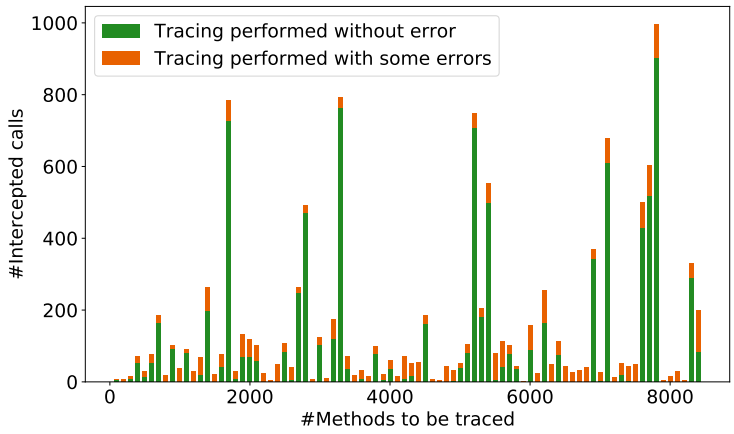
ODILE. We consider two configurations, representing the best case (noted *short* in Fig. 4.6b) and the worst case (noted *long* in Fig. 4.6b) of using ODILE. The *short* configuration represents the case where ODILE never interferes during a function call. This represents the shortest path in the execution of ODILE. The *long* configuration represents the case where ODILE intercepts and traces all calls, representing the longest execution path that ODILE can lead. We can see from Fig. 4.6b that ODILE overhead is relatively stable. At worst the slowdown is about 60%. In some cases, the impact of ODILE is so negligible that the first activity shows up faster than the baseline, meaning that we are in the margin of error.

Scalability and resource consumption

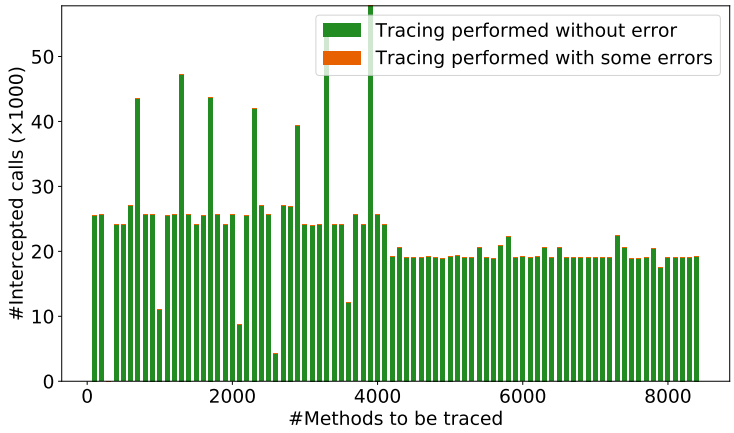
Scalability. The evaluation methodology is as follows. We firstly build a set of 14,000 distinct function signatures that we want to monitor during the execution of the app. Then we randomly select n functions that we use as input for ODILE and FRIDA. We repeat the experiment while varying n. We compare ODILE with FRIDA. We perform the experiments on all apps but Fig. 4.7 only presents the results for DualOps app for illustration. The results obtained with other apps are similar. We can see that FRIDA is still unable to intercept large number of calls (the average number of intercepted calls is 140, see Fig. 4.7a) because it crashes early. We observe no issue with ODILE (the average number of intercepted calls is 23k calls, see Fig. 4.7b), which works regardless the number of calls.

Resource consumption. We collect memory and CPU consumption during the previous experiments. We do not compare ODILE with FRIDA in this experiment because the latter crashes early. Fig. 4.8a presents the CPU consumption for an execution. In fact, we observed no CPU variation related to the number of functions to trace. Fig. 4.8a shows that ODILE incurs almost no CPU overhead. Concerning the memory, Fig. 4.8b shows the results for all experiments: each point corresponds to the maximum memory consumption measured during the execution of ODILE with a given number of functions to monitor. We can observe almost no overhead introduced by ODILE related to the variation of the number of functions to trace: the reader should compare the value of y at $x = 0$ with the value of y for the other values of x [5].

[5] we removed the point (300 methods ,155.324 MB), considering it as a outlier one

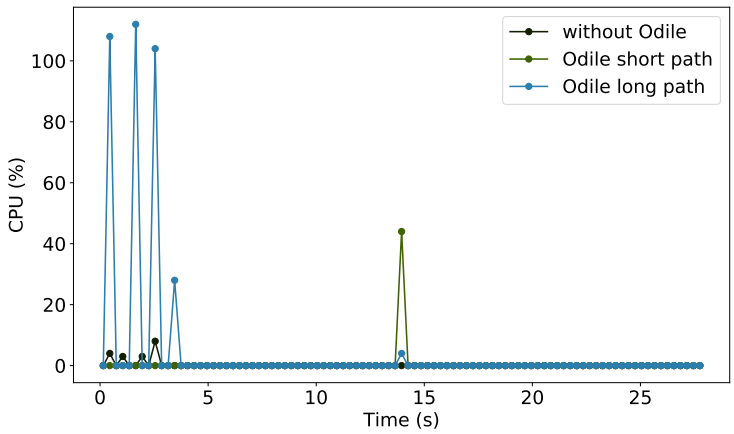


(a) Frida

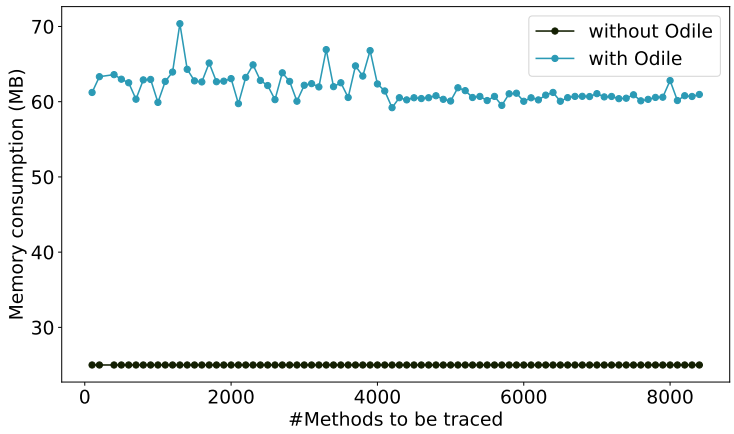


(b) Odile

Figure 4.7: Odile scalability on DualOps app, compared with Frida. The green color represents the number of call interception realized by Frida without error while and the orange one represents interceptions which lead to errors.



(a) CPU



(b) Memory

Figure 4.8: Odile CPU and memory consumption for DualOps app.

4.7 Related work

State-of-the-art tracing tools require inherently to be able to instrument Android applications, and so far code instrumentation requires most often either to customize the underlying Android system, or to jailbreak/root the Android devices [39, 27, 26, 24, 92, 103, 71, 37, 95, 78, 94, 76, 32, 55]. Some of them are dedicated to QEMU emulator such as Droidscope [95], Copperdroid [78], Malton [94] and Mobile-sandbox [76]. Particularly, Droidscope and Copperdroid are designed to especially trace system calls performed by the Android system, whereas Malton and Mobile-sandbox are designed to trace call graph of running Android applications. Anyway, none of the 4 aforementionned works have been designed to be executed on off-the-shelf Android devices as opposed to IARMDroid [39], Appguard [27, 26], Artist [24], and Aurasium [92], DroidBox [55]. However, none of the latters can run on unaltered/non-jailbroken Android devices. The only exceptions are Crowdroid [32], DroidTrace [102], and Updroid [79]. Crowdroid leverages *strace* to monitor system calls, whereas DroidTrace uses *ptrace*. However, nowadays, *strace*, nor *ptrace* are available anymore on the Android system for security reasons.

As far of our knowledge [79] is the only monitoring system which has been built for end users and off-the-shelf devices. The authors proposed to monitor intents rather than directly tracing app calls. Nevertheless, the evaluations performed by the authors showed that this approach leads to a significant number of false positive and false negative. Building a behavioral analysis tool which is based on app generated calls is the holy grail, thus motivating ODILE.

4.8 Conclusion

We introduced ODILE, the first lightweight, scalable, and interoperable tracer for off-the-shelf x86 and ARM Android devices. We discussed our *delegated instrumentation* tracing approach, which leverages Android's instrumentation module. ODILE is the building block for app behavior analysis on end user phones to accurately identify malware. We demonstrated the effectiveness of ODILE in tracing various app types on Samsung Galaxy A7 2017. We showed how much ODILE outperforms FRIDA, the state-of-the-art tool in the domain. The results also showed that ODILE is scalable and incurs no overhead.

Chapter 5

Conclusion

5.1 Conclusion

We have proposed solutions to resolve some issues in the war against publishing of illegitimate applications on mobile markets. This chapter provides a conclusion to this work, and outlines future prospects.

We have seen in the context of our work that solutions for detecting malicious mobile applications, static as well as dynamic, are subject to the problems of accuracy and evolution. This is due to techniques implemented in malwares, which are becoming increasingly sophisticated. It is also due to the fact that solutions against malwares must be integrated into the application deployment process, which implies a certain scalability. Our contributions are mainly aimed at solving the last two issues, namely evolution and scalability.

We have identified a brand new technique for deploying malicious applications, consisting for an attacker to have his application mimicking a web service or a known company whose mobile version does not yet exist. Then we have proposed a solution called IMAD (Illegitimate Mobile App Detector) which, easily deployable on the commodities desktop at store level, detects with an accuracy of 80% the illegitimate applications protecting both big, medium and small-sized companies from the usurpation of their graphic charter.

Moreover, we have noticed the difficulty with which dynamic crowd-based analysis techniques, a current trend in the research field, are hardly advancing. Since the goal of these techniques is to maximize the number of people who can test the applications on their phones at any given time, it must be possible to implement them on most phones. However, this is not the case because they only work on "rooted" phones limiting the number of phones

and users, and even if it were, they are imprecise. We have proposed Odile, which is based on indirect code injection from inside of the application to be analyzed and through a preexisting module of the Android system. We were able to install it on the x86 architectures commonly used by emulators and on the ARM64 architectures used by most phones, the evaluation example being a Samsung Galaxy A7. Odile competes in terms of performance with the most advanced existing solutions in the field, and also in terms of accuracy because it can detect all the APIs commonly considered critical by the literature.

Finally, the detection of malicious applications raises challenges that are strongly associated with the evolution of the mobile ecosystem in its environment. On the one hand, hackers implement new techniques for deploying and executing their loads that must be identified and addressed with the help of all the tools available in the computing world such as open source data sources and artificial intelligence algorithms. On the other hand, the proposed solutions are not always easy to integrate into the ecosystem because of the technical limitations related to operating systems that will have to be circumvented by strong methods such as hacking, but a hacking that is very benign for user protection.

Bibliography

[1] 10 exclusive f-droid apps you can't get on google play store. https://www.makeuseof.com/tag/exclusive-f-droid-apps/. Visited on October 2020.

[2] Android runs 52 https://www.theguardian.com/technology/2011/nov/15/android-runs-most-smatphones-sold. visited on January 2021.

[3] App statistics on mobile market places. https://www.businessofapps.com/data/app-statistics/.

[4] Dalvik bytecode. https://source.android.com/devices/tech/dalvik/dalvik-bytecode. Visited on June 2020.

[5] Database of 298 android malwares. https://github.com/ashishb/android-malware. Visited on September 2020.

[6] Duktape. https://duktape.org/. Visited on January 2020.

[7] Frida url. https://frida.re/. Accessed: 2019-10-14.

[8] Google's bouncer for android shows malware apps the door. https://mashable.com/2012/02/02/google-bouncer-for-android/?europe=true. Visited on April 2020.

[9] jadx - dex to java decompiler. https://github.com/skylot/jadx. Visited on January 2021.

[10] libffii. https://github.com/libffi/libffi. Visited on December 2020.

[11] Mobile operating system market share worldwide. https://developer.android.com/studio/test/monkey. Visited on June 2019.

[12] Number of mobile app users. https://www.my-business-plan.fr/chiffres-application.

[13] Ransomware on mobile devices: knock-knock-block. https://www.kaspersky.com/blog/mobile-ransomware-2016/12491/. visited on September 2018.

[14] Simpletexteditor use to evaluate the effectiveness of odile. https://f-droid.org/en/packages/com.maxistar.textpad/.

[15] Ui/application exerciser monkey. http://gs.statcounter.com/os-market-share/mobile/worldwide. Visited on June 2019.

[16] Allix, K., Bissyandé, T. F., Klein, J., and Traon, Y. L. Androzoo: Collecting millions of android apps for the research community. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)* (2016), pp. 468–471.

[17] androguard. androguard. https://github.com/androguard/androguard.

[18] appstoremetrics. The lack of app store metrics. https://www.ben-evans.com/benedictevans/2015/6/13/the-lack-of-app-store-metrics.

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

[20] Arzt, S., Rasthofer, S., and Bodden, E. The soot-based toolchain for analyzing android apps. In *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems* (Piscataway, NJ, USA, 2017), MOBILESoft '17, IEEE Press, pp. 13–24.

[21] aspell. The metaphone distance. http://aspell.net/metaphone/.

[22] Au, K. W. Y., Zhou, Y. F., Huang, Z., and Lie, D. Pscout: Analyzing the android permission specification. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (New York, NY, USA, 2012), CCS '12, Association for Computing Machinery, p. 217–228.

[23] Azim, T., and Neamtiu, I. Targeted and depth-first exploration for systematic testing of android apps. *SIGPLAN Not. 48*, 10 (Oct. 2013), 641–660.

[24] Backes, M., Bugiel, S., Schranz, O., von Styp-Rekowsky, P., and Weisgerber, S. Artist: The android runtime instrumentation and

security toolkit. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)* (2017), IEEE, pp. 481–495.

[25] Backes, M., Gerling, S., Hammer, C., Maffei, M., and Styp-Rekowsky, P. Appguard – fine-grained policy enforcement for untrusted android applications. vol. 8247.

[26] Backes, M., Gerling, S., Hammer, C., Maffei, M., and von Styp-Rekowsky, P. Appguard - enforcing user requirements on android apps. In *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings* (2013), N. Piterman and S. A. Smolka, Eds., vol. 7795 of *Lecture Notes in Computer Science*, Springer, pp. 543–548.

[27] Backes, M., Gerling, S., Hammer, C., Maffei, M., and von Styp-Rekowsky, P. Appguard - fine-grained policy enforcement for untrusted android applications. In *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* (2013), J. García-Alfaro, G. V. Lioudakis, N. Cuppens-Boulahia, S. N. Foley, and W. M. Fitzgerald, Eds., vol. 8247 of *Lecture Notes in Computer Science*, Springer, pp. 213–231.

[28] Balabantaray, R. C., Sarma, C., and Jha, M. Document clustering using k-means and k-medoids. *arXiv preprint arXiv:1502.07938* (2015).

[29] Bianchi, A., Corbetta, J., Invernizzi, L., Fratantonio, Y., Kruegel, C., and Vigna, G. What the app is that? deception and countermeasures in the android user interface. In *2015 IEEE Symposium on Security and Privacy* (2015), IEEE, pp. 931–948.

[30] Bing. Bing search engine. https://www.bing.com/.

[31] Bottazzi, G., Casalicchio, E., Cingolani, D., Marturana, F., and Piu, M. Mp-shield: A framework for phishing detection in mobile devices. In *2015 IEEE International Conference on Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing* (2015), IEEE, pp. 1977–1983.

[32] Burguera, I., Zurutuza, U., and Nadjm-Tehrani, S. Crowdroid: Behavior-based malware detection system for android. pp. 15–26.

[33] Chen, K., Wang, P., Lee, Y., Wang, X., Zhang, N., Huang, H., Zou, W., and Liu, P. Finding unknown malice in 10 seconds: Mass vetting for new threats at the google-play scale. In *24th {USENIX} Security Symposium ({USENIX} Security 15)* (2015), pp. 659–674.

[34] Cheng, M.-M., Mitra, N. J., Huang, X., Torr, P. H., and Hu, S.-M. Global contrast based salient region detection. *IEEE transactions on pattern analysis and machine intelligence 37*, 3 (2014), 569–582.

[35] connortumbleson. The apktool used to extract datas from apks. http://connortumbleson.com/2015/10/12/apktool-v2-0-2-released/.

[36] contagioMobile. contagiomobile blog post. http://contagiominidump.blogspot.fr/.

[37] Costamagna, V., and Zheng, C. Artdroid: A virtual-method hooking framework on android art runtime. In *IMPS@ ESSoS* (2016), pp. 20–28.

[38] David, A. Vassilvitskii s.: K-means++: The advantages of careful seeding. In *18th annual ACM-SIAM symposium on Discrete algorithms (SODA), New Orleans, Louisiana* (2007), pp. 1027–1035.

[39] Davis, B., Sanders, B., Khodaverdian, A., and Chen, H. I-arm-droid: A rewriting framework for in-app reference monitors for android applications. *Mobile Security Technologies 2012*, 2 (2012), 1–7.

[40] dBPedia. The dbpedia web site. wiki.dbpedia.org.

[41] dBPedia. World intellectual property organization. www.wipo.int.

[42] easychair. Easychair website. http://www.easychair.org/.

[43] Enck, W., Gilbert, P., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P., and Sheth, A. N. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2010), OSDI'10, USENIX Association, pp. 393–407.

[44] Forensics. Current android malware. https://forensics.spreitzenbarth.de/android-malware/.

[45] Fratantonio, Y., Qian, C., Chung, S. P., and Lee, W. Cloak and dagger: From two permissions to complete control of the ui feedback loop. In *2017 IEEE Symposium on Security and Privacy (SP)* (2017), pp. 1041–1057.

[46] Garfinkel, T., and Rosenblum, M. A virtual machine introspection based architecture for intrusion detection. *NDSS 3* (05 2003).

[47] Gianazza, A., Maggi, F., Fattori, A., Cavallaro, L., and Zanero, S. Puppetdroid: A user-centric UI exerciser for automatic dynamic analysis of similar android applications. *CoRR abs/1402.4826* (2014).

[48] Gianazza, A., Maggi, F., Fattori, A., Cavallaro, L., and Zanero, S. Puppetdroid: A user-centric UI exerciser for automatic dynamic analysis of similar android applications. *CoRR abs/1402.4826* (2014).

[49] Google. The google custum search engine. https://cse.google.com/cse/all.

[50] Google. Google search quality evaluation. http://link.fobshanghai.com/download/googlesearchqualityevaluatorguidelines.pdf.

[51] Guan, Q., Huang, H., Luo, W., and Zhu, S. Semantics-based repackaging detection for mobile apps. In *Proceedings of the 8th International Symposium on Engineering Secure Software and Systems - Volume 9639* (Berlin, Heidelberg, 2016), ESSoS 2016, Springer-Verlag, p. 89–105.

[52] Hanna, S., Huang, L., Wu, E., Li, S., Chen, C., and Song, D. Juxtapp: A scalable system for detecting code reuse among android applications. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment* (2012), Springer, pp. 62–81.

[53] Hyrynsalmi, S., Mäkilä, T., Järvi, A., Suominen, A., Seppänen, M., and Knuutila, T. App store, marketplace, play! an analysis of multi-homing in mobile software ecosystems. *Jansen, Slinger* (2012), 59–72.

[54] Jain, A. K. Data clustering: 50 years beyond k-means. *Pattern recognition letters 31*, 8 (2010), 651–666.

[55] Lantz, P. Droidbox: An android application sandbox for dynamic analysis. In *Master's Thesis at Department of Electrical and Information Technology* (2011).

[56] Li, L., Bissyandé, T. F., Papadakis, M., Rasthofer, S., Bartel, A., Octeau, D., Klein, J., and Traon, L. Static analysis of android apps: A systematic literature review. *Information and Software Technology 88* (2017), 67 – 95.

[57] Lindorfer, M. Andrubis: A tool for analyzing unknown android applications, 2012.

[58] Lindorfer, M., Neugschwandtner, M., and Platzer, C. MARVIN: efficient and comprehensive mobile app classification through static and dynamic analysis. In *39th IEEE Annual Computer Software and Applications Conference, COMPSAC 2015, Taichung, Taiwan, July 1-5, 2015. Volume 2* (2015), S. I. Ahamed, C. K. Chang, W. C. Chu, I. Crnkovic, P. Hsiung, G. Huang, and J. Yang, Eds., IEEE Computer Society, pp. 422–433.

[59] Lindorfer, M., Neugschwandtner, M., Weichselbaum, L., Fratantonio, Y., van der Veen, V., and Platzer, C. ANDRUBIS - 1, 000, 000 apps later: A view on current android malware behaviors. In *BADGERS@ESORICS 2014, Wroclaw, Poland* (2014), IEEE, pp. 3–17.

[60] M. K. Alzaylaee, S. Y. Yerima, S. S. Dynalog: an automated dynamic analysis framework for characterizing android applications. https://doi.org/10.1109/CyberSecPODS.2016.7502337, 2016.

[61] Mahmood, R., Mirzaei, N., and Malek, S. Evodroid: Segmented evolutionary testing of android apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (New York, NY, USA, 2014), FSE 2014, Association for Computing Machinery, p. 599–609.

[62] Malisa, L., Kostiainen, K., and Capkun, S. Detecting mobile application spoofing attacks by leveraging user visual similarity perception. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy* (New York, NY, USA, 2017), CODASPY '17, Association for Computing Machinery, p. 289–300.

[63] Malisa, L., Kostiainen, K., and Capkun, S. Detecting mobile application spoofing attacks by leveraging user visual similarity perception. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy* (2017), pp. 289–300.

[64] Mani, I., Yeh, A., and Condon, S. Learning to match names across languages. In *Multi-source, Multilingual Information Extraction and Summarization*. Springer, 2013, pp. 53–71.

[65] Marforio, C., Masti, R. J., Soriente, C., Kostiainen, K., and Capkun, S. Personalized security indicators to detect application phishing attacks in mobile platforms. *arXiv preprint arXiv:1502.06824* (2015).

[66] Microsoft. Microsoft azure web site. http://azure.microsoft.com.

[67] Oberheide, J., and Miller, C. Dissecting the android bouncer. *SummerCon2012, New York 95* (2012), 110.

[68] Orange. Orange bank announcement. https://www.lesechos.fr/finance-marches/banque-assurances/0211888727149-orange-bank-lancement-prevu-mi-mai-2073226.php.

[69] Pandita, R., Xiao, X., Yang, W., Enck, W., and Xie, T. WHYPER: Towards automating risk assessment of mobile applications. In *22nd USENIX Security Symposium (USENIX Security 13)* (Washington, D.C., Aug. 2013), USENIX Association, pp. 527–542.

[70] pocketgamer. App-store submissions metrics. http://www.pocketgamer.biz/metrics/app-store/submissions/.

[71] Qiu, L., Zhang, Z., Shen, Z., and Sun, G. Apptrace: Dynamic trace on android devices. In *2015 IEEE International Conference on Communications (ICC)* (June 2015), pp. 7145–7150.

[72] quora user, A. How many new apps are added to google play everyday. https://www.quora.com/How-many-new-apps-are-added-to-Google-Play-everyday.

[73] Shao, Y., Luo, X., Qian, C., Zhu, P., and Zhang, L. Towards a scalable resource-driven approach for detecting repackaged android applications. pp. 56–65.

[74] Smith, R. An overview of the tesseract ocr engine. In *Ninth international conference on document analysis and recognition (ICDAR 2007)* (2007), vol. 2, IEEE, pp. 629–633.

[75] Soh, C., Tan, H. B. K., Arnatovich, Y. L., and Wang, L. Detecting clones in android applications through analyzing user interfaces. In *2015 IEEE 23rd international conference on program comprehension* (2015), IEEE, pp. 163–173.

[76] Spreitzenbarth, M., Freiling, F., Echtler, F., Schreck, T., and Hoffmann, J. Mobile-sandbox: having a deeper look into android applications. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing* (2013), ACM, pp. 1808–1815.

[77] Tam, K., Feizollah, A., Anuar, N., Salleh, R., and Cavallaro, L. The evolution of android malware and android analysis techniques. *ACM Computing Surveys 49* (01 2017), 1–41.

[78] Tam, K., Khan, S. J., Fattori, A., and Cavallaro, L. Copperdroid: Automatic reconstruction of android malware behaviors. In *Ndss* (2015).

[79] Tang, X., Lin, Y., Wu, D., and Gao, D. Towards dynamically monitoring android applications on non-rooted devices in the wild. In *Proceedings of the 11th ACM Conference on Security & Privacy in Wireless and Mobile Networks* (New York, NY, USA, 2018), WiSec '18, ACM, pp. 212–223.

[80] Tata, S., and Patel, J. M. Estimating the selectivity of tf-idf based cosine similarity predicates. *ACM Sigmod Record 36*, 2 (2007), 7–12.

[81] tineye. Tineye. https://www.tineye.com/.

[82] trendmicro. Wp fake apps. http://www.trendmicro.com/cloud-content/us/pdfs/security-intelligence/white-papers/wp-fake-apps.pdf.

[83] Unuchek, R. Rooting pokémons in google play store. https://securelist.com/rooting-pokemons-in-google-play-store/76081/.

[84] Wapet, P. L., Tchana, A., Tran, G. S., and Hagimont, D. Preventing the propagation of a new kind of illegitimate apps. *Future Gener. Comput. Syst. 94* (2019), 368–380.

[85] Wong, M. Y., and Lie, D. Intellidroid: A targeted input generator for the dynamic analysis of android malware. In *NDSS* (2016), vol. 16, pp. 21–24.

[86] Wong, M. Y., and Lie, D. Intellidroid: A targeted input generator for the dynamic analysis of android malware. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016* (2016).

[87] Wong, M. Y., and Lie, D. Tackling runtime-based obfuscation in android with tiro. In *Proceedings of the 27th USENIX Conference on Security Symposium* (Berkeley, CA, USA, 2018), SEC'18, USENIX Association, pp. 1247–1262.

[88] Wu, D., Mao, C., Wei, T., Lee, H., and Wu, K. Droidmat: Android malware detection through manifest and api calls tracing. In *2012 Seventh Asia Joint Conference on Information Security* (2012), pp. 62–69.

[89] Wu, L., Du, X., and Wu, J. Mobifish: A lightweight anti-phishing scheme for mobile phones. In *2014 23rd International Conference on Computer Communication and Networks (ICCCN)* (2014), IEEE, pp. 1–8.

[90] Wu, L., Du, X., and Wu, J. Effective defense schemes for phishing attacks on mobile computing platforms. *IEEE Transactions on Vehicular Technology 65*, 8 (2015), 6678–6691.

[91] www.statista.com. Number of avalaible applications on google play store. https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/.

[92] Xu, R., Saïdi, H., and Anderson, R. Aurasium: Practical policy enforcement for android applications. In *21st USENIX Security Symposium (USENIX Security 12)* (Bellevue, WA, Aug. 2012), USENIX Association, pp. 539–552.

[93] Xu, Z., and Zhu, S. Abusing notification services on smartphones for phishing and spamming. In *WOOT* (2012), pp. 1–11.

[94] Xue, L., Zhou, Y., Chen, T., Luo, X., and Gu, G. Malton: Towards on-device non-invasive mobile malware analysis for art. In *26th USENIX Security Symposium (USENIX Security 17)* (2017), pp. 289–306.

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

[96] Yang, W., Xiao, X., Andow, B., Li, S., Xie, T., and Enck, W. Appcontext: Differentiating malicious and benign mobile app behaviors using context. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering* (2015), vol. 1, pp. 303–313.

[97] Yujian, L., and Bo, L. A normalized levenshtein distance metric. *IEEE transactions on pattern analysis and machine intelligence 29*, 6 (2007), 1091–1095.

[98] Zhang, F., Huang, H., Zhu, S., Wu, D., and Liu, P. Viewdroid: Towards obfuscation-resilient mobile application repackaging detection. In *Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks* (2014), pp. 25–36.

[99] Zhauniarovich, Y., Gadyatskaya, O., Crispo, B., La Spina, F., and Moser, E. Fsquadra: Fast detection of repackaged applications. In

Data and Applications Security and Privacy XXVIII (Berlin, Heidelberg, 2014), V. Atluri and G. Pernul, Eds., Springer Berlin Heidelberg, pp. 130–145.

[100] Zhauniarovich, Y., Gadyatskaya, O., Crispo, B., La Spina, F., and Moser, E. Fsquadra: Fast detection of repackaged applications. In *Data and Applications Security and Privacy XXVIII* (Berlin, Heidelberg, 2014), V. Atluri and G. Pernul, Eds., Springer Berlin Heidelberg, pp. 130–145.

[101] Zheng, C., Zhu, S., Dai, S., Gu, G., Gong, X., Han, X., and Zou, W. Smartdroid: An automatic system for revealing ui-based trigger conditions in android applications. In *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices* (New York, NY, USA, 2012), SPSM '12, Association for Computing Machinery, p. 93–104.

[102] Zheng, M., Sun, M., and Lui, J. C. S. Droidtrace: A ptrace based android dynamic analysis system with forward execution capability. In *2014 International Wireless Communications and Mobile Computing Conference (IWCMC)* (2014), pp. 128–133.

[103] Zhou, W., Wang, Z., Zhou, Y., and Jiang, X. Divilar: Diversifying intermediate language for anti-repackaging on android platform. In *Proceedings of the 4th ACM conference on Data and application security and privacy* (2014), pp. 199–210.

[104] Zhou, Y., and Jiang, X. Dissecting android malware: Characterization and evolution. In *2012 IEEE symposium on security and privacy* (2012), IEEE, pp. 95–109.

[105] Zhou, Y., and Jiang, X. Dissecting android malware: Characterization and evolution. In *2012 IEEE Symposium on Security and Privacy* (May 2012), pp. 95–109.