Highlight and execute suspicious paths in Android malware

Mourad Leslous

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE RENNES 1

THÈSE DE DOCTORAT DE

L'UNIVERSITE DE RENNES 1
COMUE UNIVERSITÉ BRETAGNE LOIRE

ECOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : *Informatique*

Par
« Mourad LESLOUS »

« Highlight and Execute Suspicious Paths in Android Malware »

Thèse présentée et soutenue à Rennes, le 18/12/2018
Unité de recherche : IRISA
Thèse N° :

Rapporteurs avant soutenance :

Yves LE TRAON	Professeur, Université du Luxembourg
Wojciech MAZURCZYK	Professeur Associé, Université de Technologie de Varsovie

Composition du Jury :

Président :	Prénom Nom	Fonction et établissement d'exercice *(à préciser après la soutenance)*
Examinateurs :	Hervé DEBAR	Professeur, Télécom Sud Paris
	Jean-Marie BONNIN	Professeur, IMT Atlantique
	Pascal BERTHOMÉ	Professeur, INSA Centre Val de Loire
Dir. de thèse :	Valérie VIET TRIEM TONG	Enseignant Chercheur, CentraleSupélec
Co-dir. de thèse :	Thomas GENET	Maître de conférences, Université de Rennes 1
Encadrant :	Jean-François LALANDE	Maître de conférences, CentraleSupélec

To my parents

Abstract

The last years have known an unprecedented growth in the use of mobile devices especially smartphones. They became omnipresent in our daily life because of the features they offer. They allow the user to install third-party apps to achieve numerous tasks. Smartphones are mostly governed by the Android operating system. It is today installed on more than 80% of the smartphones. Mobile apps collect a huge amount of data such as email addresses, contact list, geolocation, photos and bank account credentials. Consequently, Android has become a favorable target for cyber criminals. Thus, understanding the issue, i.e., how Android malware operates and how to detect it, became an important research challenge.

Android malware frequently tries to bypass static analysis using multiple techniques such as code obfuscation and dynamic code loading. To overcome these limitations, many analysis techniques have been proposed to execute the app and monitor its behavior at runtime. Nevertheless, malware developers use time and logic bombs to prevent the malicious code from executing except under certain circumstances. Therefore, more actions are needed to trigger it and monitor its behavior. Recent approaches try to automatically characterize the malicious behavior by identifying the most suspicious locations in the code and forcing them to execute. They strongly rely on the computation of application global control flow graphs (CFGs). However, these CFGs are incomplete because they do not take into consideration all types of execution paths. These approaches solely analyze the application code and miss the execution paths that occur when the application calls a framework method that in turn calls another application method.

We propose in this dissertation a tool, GPFinder, that automatically exhibits execution paths towards suspicious locations in the code by computing global CFGs that include edges representing explicit and implicit interprocedural calls. It also gives key information about the analyzed application in order to understand how the suspicious code was injected into the application. To validate our approach, we use GPFinder to

study a collection of 14,224 malware samples, and we evaluate that 72.69% of the samples have at least one suspicious code location which is only reachable through implicit calls.

Triggering approaches mainly use one of the following strategies to run a specific portion of the application's code: the first approach heavily modifies the app to launch the targeted code without keeping the original behavioral context. The second approach generates the input to force the execution flow to take the desired path without modifying the app's code. However, it is sometimes hard to launch a specific code location just by fuzzing the input. For instance, when the application performs a hash on the input data and compares the result to a fixed string to decide which branch of the condition to take, the fuzzing program should reverse the hashing function, which is obviously a hard problem.

We propose in this dissertation a tool, TriggerDroid, that has a twofold goal: force the execution of the suspicious code and keep its context close to the original one. It crafts the required framework events to launch the right app component and satisfies the necessary triggering conditions to take the desired execution path. To validate our approach, we led an experiment on a dataset of 135 malware samples from 71 different families. Results show that our approach needs more refinement and adaptation to handle special cases due to the highly diverse malware dataset that we analyzed.

Up-to-date and well documented malware datasets are crucial to design better security approaches and validate new tools. There exist a handful Android malware datasets that are dedicated for the research community. However, they rarely detail the working of their samples. In this dissertation, we give a feedback on the experiments we led on different malware datasets, and we explain our experimental process. Finally, we present the Kharon dataset, a collection of well documented Android malware that can be used to understand the malware landscape.

Résumé

Les dernières années ont connu une croissance sans précédent dans l'utilisation des appareils mobiles, plus particulièrement les smartphones. Ils sont devenus omniprésents dans notre vie quotidienne à cause des options qu'ils proposent. Ils permettent à l'utilisateur d'installer des applications tierces pour accomplir plusieurs tâches. Aujourd'hui, Android est installé sur plus de 80% des smartphones. Les applications mobiles recueillent une grande quantité d'informations, telles que les adresses mail, la liste des contacts, la géolocalisation, les photos et les identifiants bancaires. Par conséquent, Android est devenu une cible préférée des cybercriminels. Comprendre le fonctionnement des malwares et comment les détecter est devenu un défi de recherche important.

Les malwares Android tentent souvent d'échapper à l'analyse statique en utilisant des techniques telles que l'obfuscation et le chargement dynamique du code. Pour surmonter cela, des approches d'analyse ont été proposées pour exécuter l'application et surveiller son comportement. Néanmoins, les développeurs des malwares utilisent des bombes temporelles et logiques pour empêcher le code malveillant d'être exécuté sauf dans certaines circonstances. Par conséquent, plus d'actions sont requises pour déclencher et surveiller leurs comportements. Des approches récentes tentent de caractériser automatiquement le comportement malveillant en identifiant les endroits du code les plus suspicieux et en forçant leur exécution. Elles se basent sur le calcul des graphes de flot de contrôle (CFG). Cependant, ces CFG sont incomplets car ils ne prennent pas en considération tous les types de chemins d'exécution. Ces approches analysent seulement le code d'application et ratent les chemins d'exécution générés quand l'application appelle une méthode du framework, qui appelle à son tour une autre méthode applicative.

Nous proposons dans ce mémoire un outil, GPFinder, qui extrait automatiquement les chemins d'exécution qui mènent vers les endroits suspicieux du code, en calculant des CFG qui incluent les arcs représentant des appels interprocéduraux explicites et implicites. Il fournit aussi des informations clés sur l'application analysée afin de comprendre comment le code suspicieux a été injecté dans l'application. Pour valider notre

approche, nous utilisons GPFinder pour étudier une collection de 14224 malwares Android. Nous évaluons que 72,69% des échantillons ont au moins un endroit suspicieux du code qui n'est atteignable qu'à travers des appels implicites.

Les approches de déclenchement actuelles utilisent principalement deux stratégies pour exécuter une partie du code applicatif. La première stratégie consiste à modifier l'application excessivement pour lancer le code ciblé sans faire attention à son contexte originel. La seconde stratégie consiste à générer des entrées pour forcer le flot de contrôle à prendre le chemin désiré sans modifier le code d'application. Cependant, il est parfois difficile de lancer un endroit spécifique du code seulement en manipulant les entrées. Par exemple, quand l'application fait un hachage des données fournies en entrée et compare le résultat avec une chaîne de caractères fixe pour décider quelle branche elle doit prendre. Clairement, le programme de manipulation d'entrée devrait inverser la fonction de hachage, ce qui est presque impossible.

Nous proposons dans ce mémoire un outil, TriggerDroid, qui a deux buts : forcer l'exécution du code suspicieux et garder le contexte originel de l'application. Il fournit les événements framework requis pour lancer le bon composant et satisfait les conditions nécessaires pour prendre le chemin d'exécution désiré. Pour valider notre approche, nous avons fait une expérience sur 135 malwares Android de 71 familles différentes. Les résultats montrent que notre approche nécessite plus de raffinement et d'adaptation pour traiter les cas spéciaux dus à la grande diversité des échantillons analysés.

Avoir des collections de malwares à jour et bien documentées sont cruciales pour concevoir des meilleures approches de sécurité et pour valider les nouveaux outils. Il existe quelques collections de malwares Android dédiées à la communauté de recherche. Cependant, elles détaillent rarement le fonctionnement de leurs échantillons. Dans ce mémoire, nous fournissons un retour sur les expériences que nous avons conduites sur différentes collections, et nous expliquons notre processus expérimental. Finalement, nous présentons le dataset Kharon, une collection de malwares Android bien documentés qui peuvent être utilisés pour comprendre le panorama des malwares Android.

Résumé étendu de la thèse

Ma thèse de doctorat, commencé en septembre 2015 et qui va être soutenue en décembre 2018, a été co-dirigée par Valérie VIET TRIEM TONG (enseignant-chercheur à CentraleSupélec) et Thomas GENET (Maître de conférences à l'Université Rennes 1), et co-encadrée par Jean-François LALANDE (Maître de conférences à CentraleSupélec). Elle se propose de *"Mettre en avant et exécuter les chemins suspicieux dans les malwares Android"*. Elle a été réalisée eu sein des laboratoires de l'Institut National de Recherche en Informatique et en Automatique (INRIA).

Thématique

Cette thèse porte sur la sécurité des applications Android. L'utilisation des smartphones a explosé ces dernières années. Ces appareils sont devenus omniprésents dans notre vie quotidienne. Cela peut être expliqué par la multitude des options que ces appareils proposent, notamment les applications tierces, la connexion Internet mobile et la présence d'un appareil photo. Les smartphones sont majoritairement gouvernés par le système d'exploitation Android. En fait, en 2016, Android était installé sur 87% des smartphones [1]. D'ailleurs, Android est le système d'exploitation le plus utilisé sur la planète, bien plus que les systèmes d'exploitation des ordinateurs de bureau. Il est utilisé sur une variété d'appareils allant des montres connectées aux voitures, en passant par les smartphones et les télévisions. Android est supporté par plusieurs architectures et modèles de processeurs. Il est en majorité un logiciel libre et open source, dont le code est distribué sous le projet AOSP (Android Open Source Project). Les développeurs à travers le monde développent des applications pour cette plateforme et les mettent sur des marchés d'application, comme Google Play. En décembre 2017, Le nombre d'applications sur ce dernier a atteint 3,5 millions [2].

Les smartphones contiennent différents types de données numériques, comme des photos de vacances, mais aussi des mots de passe des comptes bancaires. Les smartphones Android partagent un modèle de sécurité commun. Cela inclut SELinux, l'isola-

tion des applications, les primitives cryptographiques et les mécanismes de sécurité offerts par le noyau Linux [3]. Néanmoins, les smartphones Android sont devenus une cible favorite des logiciels malveillants, encouragés par sa nature ouverte et son API riche. Pour distribuer leurs logiciels malveillants, les cybercriminels utilisent des techniques de *repackaging* : ils injectent le code malveillant dans des applications populaires légitimes et les distribuent dans des marchés d'applications alternatifs [4], [5]. Par conséquent, comprendre comment ces applications malveillantes fonctionnent et les façons de détecter et atténuer leurs attaques est devenu un défi de recherche de taille.

Pour lutter contre la prolifération des malwares Android, plusieurs approches ont été proposées. Parmi ces approches, celles qui reposent sur l'analyse statique sont limités par la difficulté du retro analyse du code [6], [7]. En effet, ces techniques souffrent de quelques limitations, particulièrement à cause des données manquantes qui ne sont disponibles qu'à l'exécution. D'ailleurs, les applications malveillantes utilisent fréquemment des techniques d'évasion d'analyse statique. Par exemple, elles utilisent *l'obfuscation* et le chargement dynamique à partir d'un serveur distant ou d'un fichier local [8], [9].

D'autres approches d'analyse de malware préfèrent observer le comportement de ces applications à l'exécution pour surmonter les limitations de l'analyse statique [10]. Cependant, les malwares Android ajoutent des contre-mesures pour échapper aux exécutions qui tentent de les analyser. Par exemple, ils emploient les conditions de déclenchement qui sont faciles à implémenter, et qui protègent le code malveillant en retardant son exécution une durée configurée au préalable ou en attendant la réception d'un SMS [11].

Appels implicites

Des approches d'analyse de malware Android récentes essayent de caractériser automatiquement le comportement malveillant en se basant sur une combinaison d'analyse statique et dynamique [10], [12]–[14]. Elles ont pour objectif de localiser, étudier et exécuter des parties spécifiques du code de l'application afin de surveiller leurs comportements. Ces approches se basent fortement sur le calcul des graphes de flot de contrôle (CFG) qui représentent tous les chemins d'exécution présents dans l'application [15]. De tels CFG ne sont utiles que s'ils sont complets, ou au moins dans ce contexte, lorsque ils contiennent les chemins d'exécution qui mènent vers les parties suspicieuses du code. Malheureusement, ces approches ne prennent pas en considération tous les types de

chemins d'exécution parce quelles n'analysent que le code de l'application. Par conséquent, elles ignorent les chemins d'exécution qui passent par le framework Android. Les applications Android utilisent fréquemment les *callbacks* pour différentes raisons, comme traiter les événements générés par l'interface graphique, les messages de diffusion ou le multithreading. À titre d'exemple, quand une application Android veut lancer une tâche de fond, elle réimplémente la classe `AsyncTack` et met le code qui exécute la tâche dans la méthode `doInBackground()`. Ensuite, quand elle veut lancer cette tâche, elle appelle la méthode `execute()` de la sous-classe qu'elle a réimplémentée d'`AsyncTack`. Dans ce cas, l'application n'appelle jamais directement la méthode `doInBackground()`, c'est le framework Android qui s'occupe de cela, ce qui explique le nom *callback*. Ainsi, après création du CFG de l'application, celui-ci sera incomplet, à cause du callback. Si un code suspicieux se trouve dans une telle méthode qui est appelée *implicitement*, il sera considéré comme inatteignable par la plupart des outils d'analyse statique car ils ne prennent pas le framework Android en considération. Dans ce manuscrit, nous confirmons que les CFG utilisés par les outils d'analyse de malware Android actuels sont incomplets car ils ne prennent pas en considération les appels de flot de contrôle implicites.

Avant de pouvoir proposer notre contribution à ce problème, nous avons besoin de souligner les propriétés du langage Java et la nature du framework Android qui rendent la détection des malwares Android difficile. Par conséquent, nous aurons besoin de concevoir un outil qui prend une application Android et génère les informations nécessaires comme les endroits suspicieux et le graphe de flot de contrôle global qui prend en considération les callbacks Java et Android.

Nous proposons dans ce manuscrit une approche qui fournit les chemins d'exécution qui mènent vers les parties suspicieuses du code d'application en calculant des CFG qui incluent les appels inter-procéduraux explicites et implicites. Nous implémentons notre approche dans un outil appelé GPFinder (*GroddDroid Path Finder*). GPfinder fournit aussi d'autres informations qui aident à comprendre le code malveillant et comment il a été inséré dans l'application.

Pour valider notre approche, nous utilisons GPFinder pour étudier une collection de 14224 applications malveillantes. Nous montrons à la suite de cette étude qu'on peut améliorer l'analyse de malware Android en incluant les appels implicites dans les CFG. Nous évaluons que 72.69% des échantillons analysés ont au moins un endroit suspicieux du code qui n'est accessible qu'à travers les appels implicites. En plus, nous examinons la structure commune des malwares Android et nous montrons leurs points d'entrée favoris.

Déclenchement du code suspicieux

Les techniques d'analyse qui se basent sur l'observation du comportement des applications à l'exécution sont confrontées à un autre type de défi : les malwares Android peuvent ne pas exposer leurs comportements malveillants sauf sous certaines circonstances [11], [16]. Le code malveilant peut rester inactif une certaine durée avant de lancer l'attaque. De plus, de multiples événements système peuvent être utilisés pour retarder l'exécution du code malveillant. Par exemple, le code malveillant peut attendre la réception d'un SMS, le redémarrage du téléphone ou d'autres événements pour lancer l'action malveillante. Une inspection manuelle s'avère utile pour analyser un échantillon et identifier les conditions d'analyse qui donnent une observation réussite à l'exécution. Néanmoins, cet effort de rétro-ingénierie ne peut pas être appliqué sur les milliers d'applications chargées quotidiennement sur les marchés d'applications en ligne. Par conséquent, il faudra d'abord déclencher automatiquement le code suspicieux pour observer son comportement à l'exécution. Cela aide à mieux comprendre les actions des malwares et constitue une première étape vers la protection des utilisateurs et leurs smartphones.

Il existe de nombreux travaux qui portent sur le déclenchement du code suspicieux des applications Android. Par exemple, Fratantonio *et al.* ont proposé une nouvelle approche statique qui aide à détecter les bombes logiques et temporelles qui peuvent être utilisées pour protéger le code malveillant [11]. Cependant, ils n'évaluent pas la performance du déclenchement réussi à l'exécution. De la même façon, un outil appelé HsoMiner a été proposé par Pan X. *et al.* pour détecter les opérations sensibles cachées en se basant sur les techniques de dissimulation qui servent à échapper aux outils d'analyse dynamique [16]. Par exemple, les malwares tentent de détecter si l'appareil sur lequel ils sont exécutés est un émulateur. HsoMiner utilisent ces propriétés pour caractériser les malwares Android, malgré le fait que certains développeurs d'applications bénignes utilisent ces techniques pour protéger leurs propriétés intellectuelles. HsoMiner ne tente pas lui aussi de déclencher ces conditions de protection du code malveillant ; il les détecte seulement. Les travaux existants qui portent sur le déclenchement des malwares Android peuvent êtres classés en deux catégories : ceux qui stimulent l'application de l'extérieur sans trop modifier son code pour préserver son intégrité [10], [12], [17], [18], et ceux qui se concentrent sur le déclenchement en modifiant le code de l'application agressivement sans préserver le contexte originel du code suspicieux [19], [20].

Le déclenchement du code suspicieux en manipulant seulement les entrées de l'application peut ne pas être facile. Par exemple, il est difficile de deviner la valeur à fournir en entrée de l'application si son *hash* est comparé avec une constante pour décider s'il faut lancer le code malveillant. D'autre part, le changement excessif du code de l'application afin de déclencher le code suspicieux peut avoir pour conséquence d'exposer un comportement irréel de l'application.

Pour surmonter les techniques de protection du code, nous allons d'abord les explorer en concevant un outil qui prend une application Android et déclenche le code suspicieux qui pourrait présenter un comportement malveillant. Cet outil a besoin d'explorer les comportements suspicieux à l'exécution pour surmonter les techniques d'évasion à l'analyse statique. Il a aussi besoin d'analyser les applications dans un temps raisonnablement cours afin qu'il soit réutilisable dans des vraies situations, par exemple pour analyser toutes les applications chargées sur un marché d'application.

Nous proposons dans ce manuscrit de déclencher le code suspicieux autant que possible en préservant le contexte originel de l'application. Notre approche utilise l'analyse statique pour calculer le graphe de flot de contrôle de l'application issu de GPFinder et extrait des chemins d'exécution vers les endroits suspicieux. Ensuite, elle trouve les conditions de déclenchement qui seront impliquées dans ces chemins. Puis, notre approche utilise les techniques de dépendance de données pour extraire les variables impliquées dans ces conditions et elle calcule leurs valeurs à l'aide d'un *SMT solver*. Nous introduisons des techniques pour choisir les endroits où les modifications seront insérées pour minimiser l'impact sur le contexte d'application. Nous combinons cela avec la stimulation extérieure de l'application. Par exemple, si l'application attend un événement système, nous le fournissons pour la lancer.

Nous implémentons notre approche dans outil appelé TriggerDroid. Il opère directement sur le bytecode de l'application et ne nécessite pas son code source. Nous menons une expérience sur un dataset de 135 applications malveillantes de 71 familles différentes [21]. Les résultats montrent que notre approche nécessite davantage de raffinement et d'adaptation pour gérer les cas spéciaux à cause de la grande diversité de notre dataset analysé.

Collection d'applications malveillantes

Des collections d'applications bien documentées sont cruciales pour concevoir des meilleures approches de sécurité et pour tester les nouveaux outils de détection. Il existe quelques collections d'applications malveillantes pour Android. Cependant, elles donnent rarement des détails sur le fonctionnement de leurs échantillons. Dans ce manuscrit, nous travaillons sur des datasets d'applications malveillantes, ce qui nous permet d'évaluer l'efficacité des approches proposées. Nous proposons aussi le dataset Kharon, une collection d'applications malveillantes bien documentées destinées à la communauté de recherche. Le dataset Kharon aide à comprendre les techniques utilisées par les applications Android pour attaquer, protéger le code malveillant et communiquer avec leurs développeurs.

Contributions

Le but final de cette thèse est d'améliorer la sécurité des appareils mobiles et la vie privée de l'utilisateur en renforçant la détection des malwares Android. Dans ce présent manuscrit, nous contribuons à exposer le code malveillant et révéler les comportements malveillants potentiels dans les applications Android. Cette thèse confirme le postulat :

> Il est possible d'exposer à l'exécution les comportements suspicieux dissimulés dans les applications Android malgré la nature événementielle du système Android et les techniques d'évasion statiques et dynamiques que les malwares Android emploient.

Nous résumons les contributions de cette thèse en ce qui suit :

- Nous explorons le paradigme des callbacks Android et les propriétés héritées du langage Java et comment cela rend l'analyse des applications Android difficile.
- Nous concevons et implémentons un outil appelé GPFINDER [22] qui prend une application Android et génère un graphe de flot de contrôle interprocedural qui prend en considération les propriétés du framework Android, et trouve les chemins d'exécution nécessaires qui mènent aux endroits suspicieux de l'application.

- Nous explorons les techniques utilisées par les malwares Android pour dissimuler le code malveillant, et comment cela peut empêcher la détection par les outils d'analyse automatiques.
- Nous concevons et implémentons un outil appelé TRIGGERDROID qui prend une application Android et qui déclenche les endroits suspicieux du code qui peuvent révéler des comportements malveillants de l'application.
- Nous travaillons sur des datasets d'applications malveillantes, ce qui nous permet d'évaluer l'efficacité des approches proposées.
- Nous proposons Kharon dataset, une collection d'applications malveillantes bien documentées destinées à la communauté de recherche [23].

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

The last years have known an unprecedented growth in the use of mobile devices especially smartphones; they become omnipresent in our daily life. One of the main reasons that can explain this, is the number of features that mobile devices offer such as touch screens and ubiquitous high-speed access to the Internet, in addition to the capabilities they have such as GPS, compass, accelerometer and gyroscope.

Most smartphones permit the user to install third-party software components called applications; or apps for short. They allow the user to achieve virtually every daily need like searching the web, emailing, gaming, chatting with other people, taking photos and publishing them in social media, checking the weather forecast, planning trips and making bank transactions, in addition to the basic phone operations like making calls and exchanging SMSs. Combining all these functionalities, smartphones can be seen as replacements for many traditional tools such as alarms, cameras, paper calendars and newspapers, music players, radios, calculators, CDs and video tapes. Apps are distributed in online *markets* or *stores*. Some of them are official like Google Play and Apple App Store, while others are third-party. Third-party app stores can be general like Amazone Appstore, but they can also target specific segments of the market like by country. For example Tencent MyApp is a popular app store in China and SK T-Store is a key app store in order to access the Korean market.

Smartphones carry much information about the user. Apps collect data like email addresses, contact list, geolocation, photos and bank account credentials for benign purposes like subscribing and accessing different services, and sometimes for questionable purposes such as displaying targeted ads and selling the user's information to their partners. Cross referencing information collected about the user may also deduce even more information that he does not necessary agree to share with these apps. For instance, apps can deduce the locations of the user's home and work place from the access points he

connects to during the work hours versus in the evening.

Smartphones are mostly governed by the Android operating system. For instance, Android dominated 87% of the smartphone market in the second quarter of 2016 [1]. In fact, Android is the most installed operating system on the planet, even more than desktop OSs. The main competitor OS of Android is IOS, the Apple mobile operating system. There exist also other mobile OSs, but most of them have been discontinued like Symbian, BlackBerry OS, Firefox Os and Ubuntu Touch. Android is installed on a wide variety of hardware such as connected watches, cars and TVs, but mainly on smartphones. It is mostly free and open source and it is distributed under the Android Open Source Project (AOSP). The wide adoption of Android today is due to its open source nature, but also because of its support of a multitude of hardware thanks to the underlying Linux kernel. Developers from around the world are uploading applications to the Google Play store. For instance, in December 2017, it has reached 3.5 million applications [2].

Devices that run Android share a common platform-level security model that includes SELinux, application isolation, cryptographic primitives and the security mechanisms offered by the Linux kernel. In addition, many Android devices include advanced locking mechanisms, like fingerprint, iris scanning, and face recognition.

Naturally, Android has become a favorable target for cyber criminals because of the wide range of information smartphones contain, and also due to the low cost of malware deployment on many devices. Recent reports showed that malware of numerous types are targeting Android for different purposes, like stealing information or charging the user by sending premium SMS texts [3]. Malware authors prefer the easy method of repackaging/piggybacking to distribute their code: they inject the malicious code in popular applications and redistribute them in alternative markets [4], [5]. Thus, understanding the issue, i.e, how Android malware operates and how to detect it, became an important research challenge.

To counter Android malware spread, many detection approaches were proposed. Some of them use the Android permission mechanism in order to detect malware [24], [25]. Nevertheless, recent studies showed that permissions alone are not sufficient to detect malware because app authors tend to ask for more permission than their apps really need [26], [27].

There exist other detection approaches that do more complex work basing on static analysis [6], [7]. Nevertheless, static analysis tools use code manipulation techniques that require efforts to reverse and understand the code. Often, these techniques suffer

from limitations, especially because of the missing data that is only available at runtime. For instance, Android malware frequently tries to bypass static analysis vetting using multiple techniques like code obfuscation and dynamic code loading where the malicious code is downloaded from a remote server or loaded from a local file [8], [9].

To overcome these limitations, many dynamic analysis techniques were proposed. They execute the app and monitor its behavior at runtime. Nevertheless, this approach can also suffer from other limitations. Malware developers add countermeasures to escape execution environments that may be used to detect malware. For instance, they prevent the malicious code from executing except under certain circumstances such as in a specific country, or after a certain system event, a reception of a command from a remote server, or after a specific duration [11].

Indeed, it is difficult to observe the behavior of a suspicious code if it is protected by logic bombs. Therefore, more actions are needed to trigger it and monitor its behavior. Many recent approaches try to automatically characterize the malicious behavior by combining static and dynamic analyses, where a first static analysis identifies the most suspicious locations in the code and then a particular run of the application targets the execution of the code previously identified as suspicious [12], [19], [28], [29]. In other words; they aim at locating, studying and executing specific parts of the application in order to expose any suspicious behavior. These approaches strongly rely on the computation of application global Control Flow Graphs (CFGs) that represent all execution paths in the program [15]. Such CFGs are useful only when they are complete, or at least in this context, when they contain the necessary execution paths towards the suspicious code. In this thesis, we claim that the CFGs used by the aforementioned state-of-the-art approaches are incomplete because they do not take into consideration all types of execution paths. They solely analyze the application code, which leads to miss execution paths that pass through implicit control flow calls, i.e., those that occur when the application calls a framework method that calls another application method. This callback mechanism is used a lot in Android because of its event-driven nature.

We propose in this dissertation an approach to build control flow graphs for Android applications taking into consideration explicit but also implicit interprocedural calls. We show through our experiments that many applications have suspicious code that is reachable only through implicit calls.

Triggering approaches mainly use one of the following strategies to run a specific portion of the application's code: they either heavily modify the app to launch the targeted code, some times by extracting the targeted code and launching it in a dummy

program [20], or they keep the application code unmodified and use some fuzzing methods on the app input to force the execution flow to take the desired path [12]. The first method that heavily modifies the app has a major drawback; the targeted code can lose its original context, and therefore, it does not represent anymore the malicious behavior originally embedded in the app. The second triggering method tries to conserve the application code integrity but it is sometimes hard to launch a specific code location just by fuzzing the input. For instance, when the application performs a hash on the input data and compares the result to a fixed string to decide which branch of the condition to take, the fuzzing program should reverse the hash function, which is virtually an impossible problem.

To reveal the malicious behaviors of Android applications, we propose in this dissertation an approach that forces malware to launch the targeted locations of its code by combining the two aforementioned triggering strategies. The aim of our approach is twofold: force the execution of the suspicious code and keep its context close to the original one. It crafts the required framework events to launch the right app component, and satisfies the necessary triggering conditions to take the desired execution path. This approach relies on static analysis techniques that compute control flow paths, and exploits slicing methods to find the best places to modify variables involved in the triggering conditions. The experiments that we led show that our approach, while promising in suspicious code triggering, needs more refinement and adaptation to handle special cases due to the highly diverse malware dataset that we analyzed.

Up-to-date and well documented malware datasets are crucial to design better security approaches and validate new tools. There exist a handful of Android malware datasets that are dedicated for the research community. However, they rarely detail the working of their samples. In addition, there exist hundreds of articles online explaining new discovered malware. However, they are mostly destination to the public. They are rarely detailed enough so the researcher can get a clear idea on how the malicious action is exactly launched and how it operates. In this dissertation, we give a feedback on the experiments we led on different malware datasets, and we explain our experimental process. We also present the Kharon dataset, a collection of well documented samples that can be used to understand the malware landscape.

This chapter is structured as follows: we first present our motivating example for our work and why we are targeting particular challenges that prevent malicious code from triggering in automated analysis environments. Then, we explain the dissertation goal and scope. Next, we formulate the thesis statement. And finally, we present the dissertation outline.

1.1 Motivating Example

In this section, we present an example that motivates our research work and explains two major points that we tackle in this dissertation. Listing 1.1 shows why triggering the suspicious code is important for dynamic analysis tools that detect Android malware based on its runtime behavior. The second point this listing is showing is the importance of implicit interprocedural calls to trigger the suspicious code.

In this example, the suspicious code that sends the location to a given phone number (line 31) is protected by several conditions that should be satisfied in order to launch it. In order to trigger this suspicious code, the application must receive an `SMS-RECEIVED` intent (a communication message in Android terminology), which triggers the `onReceive` method beginning at line 1. First, the application verifies if the received intent's action is really due to an SMS reception (line 3). Then, to extract the received message and sender number, the app checks if the intent's bundle is not null at line 6. Next, the app verifies if the PDU object i.e. the content of the message is not empty at line 9. At line 20, the app checks if the sender number equals a certain stored number. Finally, it verifies if the message content contains a specific command (line 23) in order to decide if the location should be sent back or not.

The conditions that protect the malicious code are of different types: some of them check an intern value (sender number, message content) and others check the event that triggered the broadcast receiver. Randomly fuzzing the application (the two parameters: context and intent) would have little chance to trigger the malicious behavior. The app checks the content of two strings (sender and message at lines 20 and 6 respectively). This makes the chance of randomly guessing the right sender number and message content minimal.

Android applications make heavy use of callbacks in Android for different reasons like UI events handling, broadcast messages handling, multithreading, etc. For example, method `doInBackground()` in Listing 1.1 is implemented by the application but invoked by the Android framework. This method does not have an explicit call inside the application code. Therefore, the CFG that is build solely by analyzing the app code may not contain an incoming control flow edge starting from the application itself and going to `doInBackground()`. This is why such methods are called callbacks. If a malicious code is located in a method which is *implicitly* called, it will be considered as unreachable by tools that analyze only the application code. Thus, we have to analyze additional code outside the application, namely the Android framework to figure out implicit calls and build a reliable CFG.

```
1  public void onReceive(Context context, Intent intent) {
2      if (intent.getAction()
3           .equals("android.provider.Telephony.SMS_RECEIVED"))
4      {
5        Bundle bundle = intent.getExtras();
6          if (bundle != null) {
7              // Get message and sender
8              Object[] pdus = (Object[]) bundle.get("pdus");
9              if (pdus.length == 0) {
10                 return;
11             }
12             SmsMessage[] messages = new SmsMessage[pdus.length];
13             StringBuilder sb = new StringBuilder();
14             for (int i = 0; i < pdus.length; i++) {
15                 messages[i] = SmsMessage.createFromPdu((byte[])
                       pdus[i]);gitageBody());
16             }
17             String num = messages[0].getOriginatingAddress();
18             String message = sb.toString();
19             // Check sender
20             if(!num.equals("987654321"))
21                 return;
22             // Check message content
23             if(message.equals("collect")){
24                MailTask mt = new MailTask("", ((Context) this))
25                mt.execute(new Integer[0]);
26             }
27         }
28     }
29 }
30 public void run(String arg) {
31    tm = getSystemService (Context.TELEPHONY_SERVICE);
32    String location = getLocationStr();
33    SmsManager sm = SmsManager.getDefault();
34    sm.sendTextMessage(num, null, location, null, null);
35 }
```

LISTING 1.1: Triggering conditions with implicit calls

1.2 Goal and Scope of this Dissertation

In this dissertation, we explore how malware benefits from the Android framework callback driven paradigm and Java language nature to make detection by automatic security analysis tools challenging. We also show the way Android malware evades detection by static vetting tools and runtime analysis environments. This dissertation confirms the thesis that it is possible to expose malicious behaviors of Android applications even when they use code protection techniques and misuse Android callback mechanism. First, we need to design a tool that takes an Android application, and generates the necessary information about it, such as the suspicious code locations and the interprocedural CFG that takes into consideration Android and Java's callblack paradigm, which is necessary to execute the targeted code. Then, to circumvent malicious code protection techniques, we need to design a tool that takes an Android application and triggers the suspicious code that may present a malicious behavior. This tool needs to expose the suspicious behaviors at runtime to circumvent static analysis evasion techniques, It also needs to analyze applications in a reasonably short time to make it reusable in real-life situations like analyzing apps uploaded to an online app market.

Note that by triggering the suspicious code in an Android application we do **not** mean to judge the maliciousness of the application. In the present dissertation, we merely expose the suspicious hidden behaviors, and it is up to a human analyst or an automatic tool to judge whether the exposed behavior is malicious or not. For example, privilege escalation code may be classified as malicious in some situations if it is used to root the phone and take control of the user's device to install harmful software. The same privilege escalation code may be classified as benign if it is present in an application designed and well-known to be used to root the phone for benign purposes like installing new ROMs or uninstalling stock apps.

1.3 Thesis Statement

This dissertation confirms the thesis that:

> It is possible to expose the hidden suspicious behaviors of Android application at runtime despite the Android framework event driven paradigm and the static and dynamic evasion techniques that Android malware tends to use.

More precisely,

TS-1 It is possible to build interprocedural control flow graphs that are useful for security analysis purposes despite the Java language callbacks and the Android framework event-driven model.

TS-2 It is possible to trigger the suspicious code in Android applications at runtime and expose their real intentions despite the static and dynamic techniques that Android malware uses to escape automatic security analysis tools.

1.4 Contributions

The final goal of this thesis is to improve the security of mobile devices and thus the privacy of users by enhancing Android malware detection. In the present dissertation, we contribute to exposing suspicious code and thus revealing potential malicious behaviors in Android applications. We resume the contributions of this dissertation in the following:

- We explore the callback driven paradigm of the Android framework and the properties inherited from the Java language, and how this makes Android application analysis challenging.
- We design and implement a tool called GPFinder [22] that takes an Android application, generates an interprocedural control flow graph taking into consideration the Android framework properties, and finds all execution paths that lead to the targeted code locations in the app [22].
- We explore the techniques used by Android malware to protect the harmful code and hide the malicious behavior, and how this can prevent detection by automatic vetting tools.
- We design and implement a tool called TriggerDroid that takes an Android application and triggers the suspicious code locations that can reveal any malicious behaviors of the application.
- We give a feedback on the experiments we led on different malware datasets, and we explain our experimental process.

- We present the Kharon dataset, a collection of well documented Android malware that can be used to understand the malware landscape.

1.5 Dissertation Outline

The goal of this thesis is to trigger and expose Android applications' suspicious behaviors. To achieve this purpose, we structured this dissertation as follows: In Chapter 2, we present in a glance the important information about the Android platform that is necessary to understand our contributions in the domain of Android malware security.

In Chapter 3, we present the most relevant works that address the subject of implicit interprocedural calls in Android, the techniques Android malware uses to escape detection, and how to expose harmful behaviors despite these protections.

In chapter 4, we highlight the importance of Control Flow Graphs in order to analyze Android application. We show that CFGs built for Android application may be incomplete because of the heavy use of Android callbacks. We also present our tool called GPFinder, which is able to build CFGs for Android applications taking into account Android callbacks and implicit interprocedural calls. It also finds execution paths towards suspicious code locations in the application, which allows launching them if needed.

In Chapter 5, we investigate the techniques of preventing the malicious code from executing in runtime analysis environments, which are used by Android malware to escape detection by automatic vetting tools. We show the importance of exposing the real intent of Android applications in order to detect malicious behaviors and protect the end user from harmful programs. We present our tool called TriggerDroid which is able to take an Android application, trigger the suspicious code locations and expose its real behavior. Unlike other existing approaches, TriggerDroid tries to expose the application malicious behavior and keep the app's original context at the same time.

In Chapter 6, we explain the dilemma of Android malware datasets between large scale collections with few details about the samples, and small collections with highly documented reports. We point out the labeling inconsistency issue and evaluate the most important existing datasets. Then, we present Kharon dataset, our well documented collection of malware. Finally, we explain why we chose certain datasets to test our security tools.

We conclude the dissertation in Chapter 7 with some perspectives.

Chapter 2

Technical Background

In this chapter we present the necessary technical background to understand the challenges facing Android malware detection, and to position our contributions that are presented later in this dissertation. We explain in the following sections the Android operating system and its architecture, Android application and their components, the mechanisms put to secure the Android ecosystem, and some principles of software analysis.

2.1 Android System

Android is an open source operating system designed initially for touch screen devices such as smartphones and tablets, and it is the most used OS on smartphones today [1]. It was created in 2003 by a company called Android inc. and was bought by Google in 2005. The first commercially available smartphone equiped with Android was announced in 2008. [1] Android was extended later for smart watches, connected objects, TVs and cars.

Since its creation, Android has known many major releases with feature introductions and bug fixes. Its versions are named by alphabetical order on deserts like Cupcake and Donut, with the last version being 9.0 codename *Pie*.

Android is based mainly on touch input like swiping, tapping, pinching and reverse pinching to interact with the underlying software. It also supports physical devices like keyboards and mice, and takes feedback from internal hardware like accelerometers and gyroscopes. It is developed using 5 programming languages: Assembly, C, C++, Java and Kotlin. The kernel is programmed mainly in C, libraries in C++, and the other

[1] beta.techcrunch.com/2008/09/23/t-mobile-officially-announces-the-g1-android-phone

applications in Java and Kotlin. In this section, we will explain the different layers of the Android stack as depicted in Figure 2.1.

Linux Kernel. Android was developed on top of a customized Linux kernel to manage processor, memory and inputs/outputs as shown in Figure 2.1. However, Android adds some low-level mechanisms to optimize it for the user, like energy consumption, performance amelioration and security enhancements. For instance, Android uses Anonymous Shared Memory (ashmem) instead of the POSIX SHM allocator because it frees shared memory units under pressure. Android also added a set of patches to the Linux kernel to manage the energy such as wakelocks. Depending on the wakelock, it prevents the system from entering energy saving mode. For example, `WAKE_LOCK_SUSPEND` prevents the system from entering suspend mode.

Memory. Android memory is generally smaller than its desktop system counterparts. Thus, Android exercises an aggressive strategy to free it. Android sets a limit on the heap size of apps. The actual size limit depends on the device and the available RAM. Applications receive an `OutOfMemoryError` when they reach their heap size limit. Basically, applications are suspended when the user changes the focus to other apps or quit them. When the memory has low free space, Android starts to kill apps in the background. The system begins by killing processes that are less used. Dues to the low RAM space, Android tries to share memory pages between processes. The latter are forked from the Zygote process that starts when the system boots, and it loads common framework code and resources to share with new processes. Sharing memory is done also by mapping most static data into a process to allow pagination out of the memory when needed.

Binder. Android does not use SysV IPC. Instead, it uses Binder for this purpose[2]. It is an IPC and RPC mechanisms similar to DBUS[3]. It follows a client-server model where the client initiates the communication and waits for a response from the server. The client has a proxy and the server has a thread pool to handle requests. Binder calls are synchronous, *i.e.,* the client waits for a response from the server to continue its execution. There is no asynchronous communication by default. In addition, data is serialized with the transmission.

[2] `https://elinux.org/Android_Binder`

[3] `https://www.freedesktop.org/wiki/Software/dbus/`

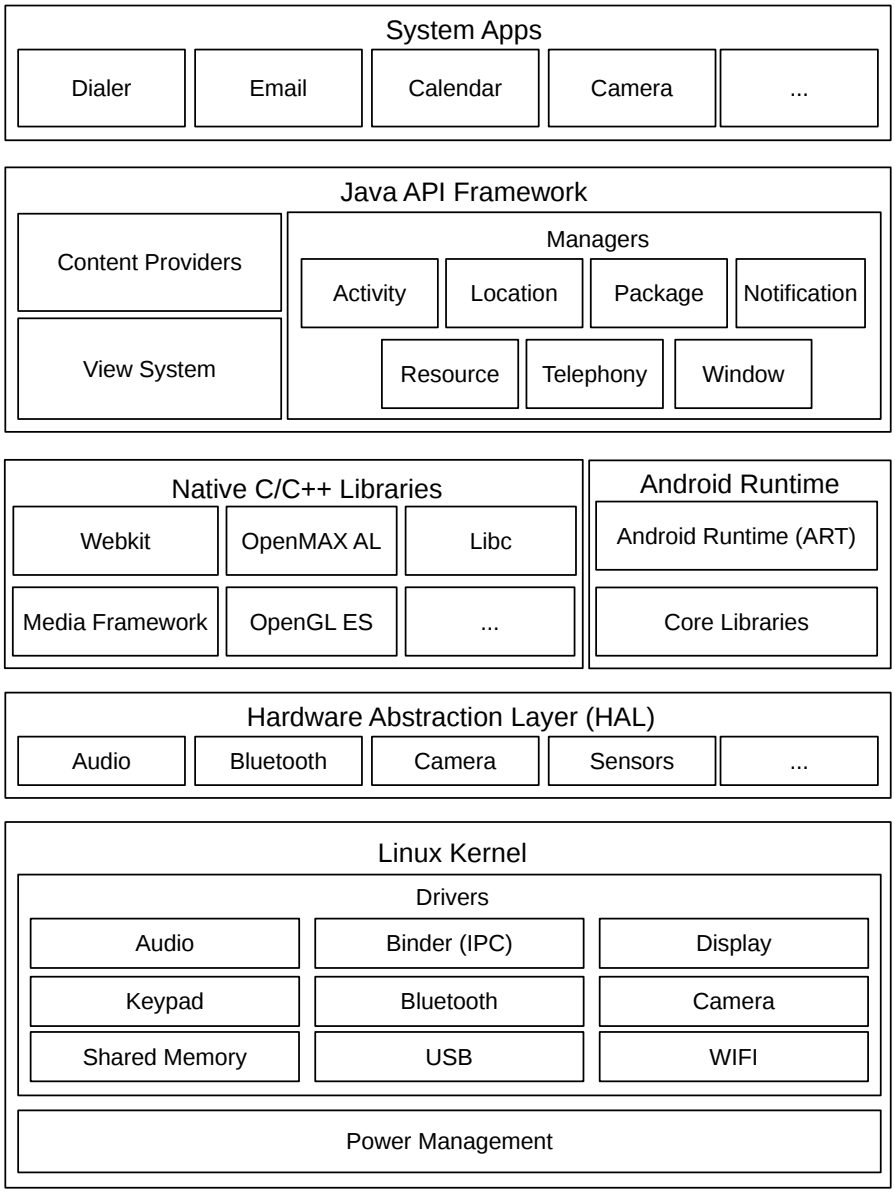

FIGURE 2.1: Android stack

Hardware Abstraction Layer. Android uses a Hardware Abstraction Layer (HAL) which provides a standard interface to interact with hardware components like Bluetooth and camera. For example, when an Android application wants to get data from a sensor, it makes a call to the corresponding function in the HAL, which in turn communicates to the sensor driver. The driver triggers the sensor and sends the data back to the HAL, so it can be passed to the Android application. This gives a complete abstraction and control over the device's sensors regardless of the vendor.

Android Runtime. Android uses a application runtime environment, ART, to execute applications which are written in java and compiled into Datvik bytecode. ART was introduced as preview in Android 4 and it replaced entirely the old Dalvik VM in Android 5.0. ART uses Ahead-Of-Time (AOT) compilation that translates application bytecode upon installation. It is basically designed to run on ARM architecture. It uses registers instead of a stack for battery saving and performance purposes.

Dalvik Bytecode. Android applications' code is compiled into Dalvik bytecode and stored in `.dex` files. They include (but not limited to) a header, sections for identifiers of string, types, method prototypes, a section for class definitions and a section for data. Dalvik bytecode approximately imitates real architecture instructions, but it operates on a register-based runtime environment. Dalvik instructions can operate on 16, 32 and 64-bit data. For example, if we take the instruction `move-wide/from16 vAA, vBBBB`:

- `move`: A base opcode that moves a register's value.
- `wide`: A suffix indicating that the opcode operates on 64-bit data.
- `from16`: A suffix indicating that source register is 16-bit long.
- `vAA`: The destination register.
- `vBBBB`: The source register.

Java API Framework. Android platform provides a wide set of APIs written in Java to enable code reuse. For example, the Resource Manager provides access for resources outside of the code like layouts and graphics. While the Activity Manager is responsible for managing the app components' lifecycle and activity stack.

Native Libraries. Many core components of the Android framework are written in C/C++ because they need native libraries to operate. It is worth noting that the core libraries do not actually performs much of the actual work and are essentially Java *wrappers* around C/C++ based libraries. Applications also can include native code to access native platform libraries.

2.2 Android Applications

Android applications are distributed in archives ending with `.apk` extension. They include a `.dex` file that contains the compiled application code, in addition to resource files, native libraries, and an application manifest. Android apps are mainly programmed in Java compiled into Dalvik bytecode, but they can contain C/C++ code via a native development kit (NDK) for high performance code like graphical rendering engines and to access low level mechanisms specific to the hardware architecture[4]. In addition, Goo language is also supported on this platform alongside with Kotlin, a newly introduced programming language.

2.2.1 APK

Android apps are distributed on online app markets like Google Play, Amazon Appstore and F-Droid under the APK (Android Package) file format. It is basically a zip file containing the compiled code and the assets needed to run the application. Note that a new format of application file called Android App Bundles was introduced in Android 9. It is basically a dynamic format where the user can download only the necessary part of the application adapted for his language, CPU architecture, screen size, etc. Android App Bundles use the same bytecode as APK, they are just a new way of delivery the application to the end user to save space.

2.2.2 Manifest

The application manifest is an XML file that describes the essential information about the app. It includes among others:

- The app's package name.

[4] `https://developer.android.com/ndk/guides/`

- App components.
- Permissions that the app needs.
- Hardware and software features the app requires.

Listing 2.1 shows an example of manifest file. For instance, it contains the app package name (line 5), the targeted SDK version and the minimum SDK version required for the app to work properly (line 7), the permission the app needs (lines 10 and 12), in this case reading and writing to the microSD card. In addition, this manifest file contains also an activity (line 22) and a service (line 29) components that we will explain later.

2.2.3 Application Components

The essential building blocks of Android apps are: activities, services, content providers, and broadcast receivers. They cooperate and may have independent lifecycles. These components constitute possible *entry points* for their application, form which the system or other apps can launch it. In this section, we present the app components and how they work.

Activities

An *activity* is basically a software component that have a graphical interface. It is used to display information and interact with the user. It is not meant to store persistent information because it can be paused at any moment if another activity is brought to foreground. Activities can take all the device screen. When another activity is launched, the current one will be pushed into a back stack. Code starting at line 22 in Listing 2.1 constitutes an example of declaring a main activity that can be launched by clicking on the application launcher icon.

To create an activity, we need to instantiate a sub-class of `Activity` and implement callbacks to be launched when the activity is created, paused, or destroyed for example. The graphical interface can be put in a separated XML file in the *resource* directory and linked to the activity, or it can be implemented in the activity code.

When an activity is launched, it can be in one of the following states:

Resumed The activity is on the first layer and it has the user focus.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
   xmlns:android="http://schemas.android.com/apk/res/android"
   android:versionCode="1" android:versionName="1.0"
   package="com.example.myapp">

   <uses-sdk android:minSdkVersion="15" android:targetSdkVersion="26" />

   <uses-permission
      android:name= "android.permission.READ_EXTERNAL_STORAGE"/>
   <uses-permission
      android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>

   <application
      android:allowBackup="true"
      android:icon="@mipmap/ic_launcher"
      android:roundIcon="@mipmap/ic_launcher_round"
      android:label="@string/app_name"
      android:supportsRtl="true"
      android:theme="@style/AppTheme">

      <activity android:name=".MainActivity">
         <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
         </intent-filter>
      </activity>

      <service android:name=".MyService" />
   </application>
</manifest>
```

LISTING 2.1: Android application's manifest example

Paused Another activity has taken the user focus. It is partially transparent and it does not cover the whole screen. It stays attached to the window manager.

Stopped The activity is on the background and not visible at all. It is still attached to the window manager.

When the state of an activity changes, the latter is notified by several callbacks that can be overridden to accomplish specific tasks. The following methods are the principle callbacks:

`onCreate()` The activity has been created freshly.

`onStart()` The activity is going to be visible.

`onResume()` The activity has become visible. It is resumed now.

`onPause()` Another activity will take the focus and this one will be paused.

`onStop()` The activity is no longer visible, it is stopped now.

`onDestroy()` The activity will be destroyed.

Figure 2.2 represents the lifecycle of the Activity, and shows transitions from a state to another and the callback methods called at each transition.

Services

A *service* is a component that executes in background to perform long-running tasks that could slow down the user experience. It does not offer a graphical interface. An application component can be attached to a service to perform a background task. A service can perform network transmission, play music, do file input/output, or interact with a *content provider*. Code at line 29 in Listing 2.1 constitutes an example of declaring a service in the manifest file.

Services have two forms:

Started When a component starts it with a `startService()` call. It can run indefinitely in order to perform its task, and stop with a `startService()` call. The service must implement `onStartCommand()` in order to be started.

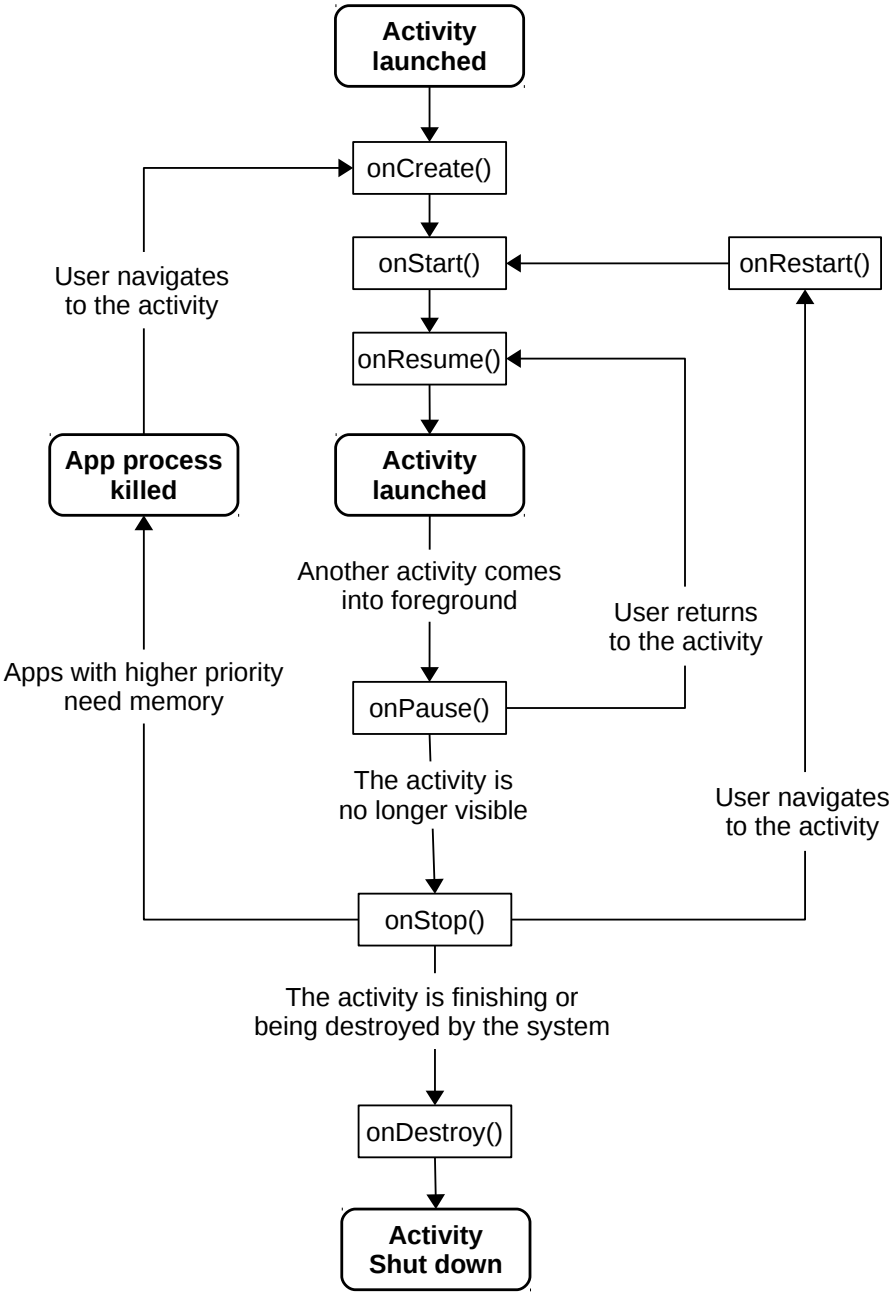

FIGURE 2.2: Activity lifecycle

Binded If a component binds with it by calling `binService()`. It offers a server-client interface to interact with it and to communicate with IPC. It runs as long as a binded component exists. The service must implement `onBind()` in order to be binded.

Content Providers

Content providers are components used to share data like files, databases and web pages with other components or applications. They manage the access to structured data and provide the mechanisms to define their security. Applications can use content providers to manage their own data. Content providers must be subclasses of `ContentProvider` and implement a set of API methods that allow other applications to perform transactions. In order to access content providers' data, applications must use a `ContentResolver` in their `Context` and communicate with the content provider as clients.

Broadcast Receivers

They are components that respond to messages broadcasted in the system announcing that some event has occurred, for example the battery is low or a picture was taken. They do no display a use interface, but they can create notifications in the status bar. Broadcast receivers are only gateways to other components, and they can not perform heavy operation. They are implemented as sub-classes of `BroadcastReceiver`, and broadcast messages are delivered as `Intents`, the Android inter-component communication message format. The app should declare the necessary permissions in the manifest to intercept the desired intents. In addition, the broadcast receiver should register for the type of intents it wants to receive in the manifest or dynamically at runtime.

2.2.4 Entry Points

Unlike C program that have one main method that constitutes the only entry point to launch the program, Android application does not have a main method and can have multiple entry points that are called by the Android framework. Entry points can be methods from the app components, such as:

`Activity.onCreate()`: To create a new activity.

`Service.onStartCommand()`: To start a service.

`Service.onBind()`: To bind with a service.

`BroadcastReceiver.onReceive()`: To receive a broadcasted Intent.

2.3 Android Security Features

Android system comes with several security features to protect the user from harmful software. For instance, it uses Security-Enhanced Linux (SELinux) to enforce the mandatory access control [30], [31]. This is applied to all processes even those with root privileges. SELinux has to modes: a *permissive mode*; where permission denials are logged by not enforced, and an *enforcing mode*; where permission denials are both logged and enforced. Android sets SELinux in enforcing mode by default.

Android implements sandboxing, where apps are executed in a *sandbox*, an isolated space that prevents the app from accessing to the rest of the system's resource it is allowed to. Android uses a permission mechanism, where apps must request them to access sensitive data like the contact list, or certain system features such as the microphone and the camera. Permissions must be declared in the manifest and should be approved by the user. For instance, `android.permission.SEND_SMS` allows the application to send SMS messages. As of Android 6.0, the user isn't notified of the permissions at installation time. Instead, he gets a pop-up at runtime to approve the permission that the app is trying to use.

In addition, Android uses some other kernel hardening features like Kernel Address Space Layout Randomization (KASLR) and post-init read-only memory[5].

2.4 Static Analysis

In this section we present some static analysis notions that are necessary to understand our contributions.

[5]Introduce post-init read-only memory, `http://linux-kernel.2935.n7.nabble.com/PATCH-v2-0-4-introduce-post-init-read-only-memory-td1247973.html`, accessed on October 26, 2018.

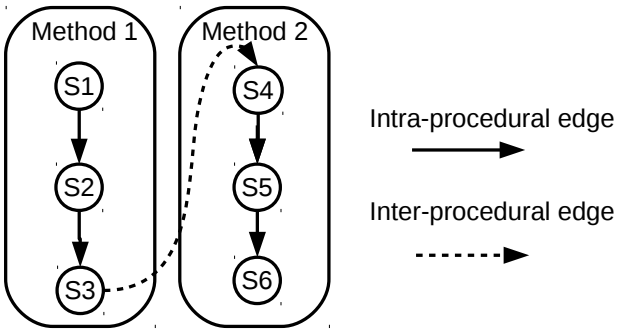

FIGURE 2.3: Inter-procedural CFG example

2.4.1 Control Flow Graph

Control flow graphs (CFGs for short) represent all the possible execution scenarios of the program, where nodes are program statements and edges are transitions between these statements [32]. There are two different types of edges in a CFG: those that represent transitions to next statements in the same method, and those that represent interprocedural calls to other methods. Interprocedural CFG (ICFG) should include both types of call. Figure 2.3 depicts an ICFG of a software of 2 methods. A call graph, on the other hand, contains only the interprocedural calls, where nodes represent program methods.

2.4.2 Data Flow Graph

Data flow graph (DFG) is a directed graph that shows data dependencies between program instructions. It is based on reaching definitions, where for a given instruction, there is an earlier instruction whose target variable can reach the given one without an intervening assignment. For example, in the following code:

```
1 S1 : y := 3
2 S2 : x := y
```

`S1` is a reaching definition for `S2`. However, in the following example:

```
1 S1 : y := 3
2 S2 : y := 4
3 S3 : x := y
```

`S1` is not a reaching definition for `S3`, because `S2` kills its reach, where the value defined in `S1` is overridden in `S2` and cannot reach `S3`.

These are the data flow equations used to find the reaching definitions for a given basic block S:

- $REACH_{in}[S] = \bigcup_{p \in pred[S]} REACH_{out}[p]$
- $REACH_{out}[S] = GEN[S] \cup (REACH_{in}[S] - KILL[S])$

The set of reaching definitions going into S is the union of all the reaching definitions from S's predecessors, $pred[S]$. $pred[S]$ consists of all basic blocks that precede S in the control flow graph. The reaching definitions coming out of S are computed by adding the reaching definitions of its predecessors except the reaching definitions whose variable is killed by S plus the new definitions generated within S.

We define the *GEN* and *KILL* sets for any given basic block S as follows:

- $GEN[d : y \leftarrow f(x_1, \cdots, x_n)] = \{d\}$, the set of definitions available in a basic block.
- $KILL[d : y \leftarrow f(x_1, \cdots, x_n)] = DEFS[y] - \{d\}$, the set of definitions killed in the basic block.

Using reaching definitions, a data dependency graph can be drawn, where nodes constitute program statements and edges link each node with its reaching definitions.

2.4.3 Reverse Engineering

According to the Institute of Electrical and Electronics Engineers (IEEE), reverse engineering is:

> "The process of analyzing a subject system to identify the system's components and their interrelationships and to create representations of the system in another form or at a higher level of abstraction." [33]

In malware analysis, reverse engineering is sometimes useful to understand the functioning of the malware in order to detect variations of it and to revert the damage done to the host machines. Reverse engineering of Android applications can be done by transforming its bytecode into a human readable representation such as Java, Smali [6], or Jimple [34].

[6] https://github.com/JesusFreke/smali/wiki

2.4.4 Analysis Tools

Analyzing Android malware is easier if we use dedicated tools and frameworks for two reasons: first, because these tools transform Dalvik bytecode to human readable formats, and second, because they implement a set of ready-to-use state-of-art static analysis algorithms to perform several tasks on the application bytecode. In this section, we present the major Android applications' static analysis tools used along this dissertation.

Intermediate Representations

Java virtual machines are stack based. The Dalvik VM, instead, uses a register-based architecture that requires fewer and more complex instructions. It is designed for systems with small memory and slower computing speed. Therefore, Dalvik bytecode is not suitable for static analysis. Several intermediate representation have been proposed to analyze Android applications. Smali[7] is an intermediate representation that is used by numerous tools such as Apktool.

Jimple is another intermediate representation. It is the main IR proposed in Soot [35]. It has a 3-address representation of the form: `x = y op z`, to keep the instruction as simple as possible. The stack is eliminated and replaced by local variables representing stack positions, that are preceded by the sign dollar. There are essentially 11 Jimple instructions, such as `assignStmt`, `invokeStmt`, `gotoStmt` and `returnStmt`. Jimple uses typed and named local variables to improve analysis and generate code. Compared to Smali, the Jimple ecosystem is supported by many research papers and by the rich Soot framework. A study shows that the Soot framework and the Jimple intermediate representation are the most used framework and IR respectively by Android static analysis works published between 2011 and 2015 [36]. For all these reasons, we chose Jimple as our main intermediate representation throughout this dissertation to perform transformation and analysis of Dalvik bytecode.

Call Graph Construction

As explained earlier, interprocedural control flow graphs are basically call graphs with nodes expended to represent method CFGs. Therefore, the main challenge in constructing ICFG is the program call graph. Since we use Soot are our basic static analyzer in this dissertation, we make use of the available callgaph construction algorithm in Soot.

[7] `https://github.com/JesusFreke/smali`

It implements several pointer analysis algorithms, such as CHA [37], RTA [38] and VTA [39]. In addition, Spark is a points-to analysis that was integrated in Soot and used to construct the callgraph and interprocedural control flow graph for application [40]. In C, points-to analysis should handle different problems, analysis of stack-directed pointers (via operator `&`) and analysis of heap-directed pointers (via operator `new`). Java point-to analysis should only handle heap-directed pointers. C programs have just one entry point method which is the `main` function. On the other hand, Java programs could have multiple entry points like initializes and finalizes, thread start methods, and methods called by reflection. In addition, Java programs can contain native methods that influence the call graph but their code is not visible to the analyzers. Spark implements subset-based [41] and equality-based [42] analyses. It is a flow-insensitive analyzer, accepts as input the Jimple 3-address representation, and supports field-sensitive and field-based analyses. Spark execution constitutes 3 phases:

Pointer assignment graph construction The graph consists of three types of nodes: allocation sites, simple variables, and field derefenrences. The analyzer starts from the main method and adds methods that are reachable from it until it gets all reachable methods.

Pointer assignment graph simplification Merging nodes that are known to have the same points-to set can help to simplify the pointer assignment graph. Especially, nodes in a strongly-connected component will have equal points-to sets, and will be merged to a single node.

Points-to set propagation Several points-to set propagation algorithms are implemented in Spark, like a naive iterative algorithm and worklist algorithm.

2.5 Conclusion

In this chapter, we explained the Android platform, Android applications, and the principals of static analysis necessary to understand the rest of this dissertation. In the next chapter we present the main state-of-the-art woks that cope with implicit calls and malware triggering in Android.

Chapter 3

State of the Art

Understanding the fundamentals of Android and its applications is useful to comprehend our contributions. Nevertheless, we do not start from the basics of Android in our contributions. Instead, we start from existing works that are trying to solve certain security issues in Android. In this chapter, we explore the threat that malware constitutes to the Android system and to the user. We discover the most relevant categories of malware that targets this platform and the techniques that were proposed in the last years to fight against the spread of malware by understanding how it works, detecting its presence, and protecting the platform and the user from its harmfulness. Then, we show the importance of control flow graphs to many Android malware analysis tools. We show that the quality of these CFGs is crucial for the security tools that are based on them. Next, we tackle the question of runtime analysis of Android malware. More precisely, why triggering the suspicious code can be crucial to detect any harmful intention that malware authors may hide in their applications? We present some of the most recent and relevant work on the subject.

3.1 Android Malware

According to Sophos, the number of Android malicious apps has risen to 3.5 billion in 2017[1]. In addition, a lot of information can be extracted from Android devices without breaking much security mechanisms [43]. For instance, shared resources between Android apps cause leaking sensitive information about the user. In addition, apps with zero permission can get sensitive information using side channels.

[1] `https://nakedsecurity.sophos.com/2017/11/07/2018-malware-forecast-the-onward-march-of-android-malware/`

In this section we will cover the most important categories of Android malware, or Potentially Harmful Application (PHA). A PHA, according to Google, is an app that puts users, user data, or devices at risk [3]. This kind of applications is different from the applications used intentionally by the user to disable some built-in security features and provide some functionality, like rooting the devices or disabling SELinux.

We would like to stress that many advanced malware families can be classified in different categories simply because they deploy several techniques to disguise the malicious code, gather useful information about the use, make money and/or leak sensitive information to remote entities. For instance, PoisonCake is a malware that sends premium SMS messages, collects phone data and uploads it to a remote server[2]. Consequently, it falls into two different malware categories; SMS fraud and spyware.

These are the most relevant Android PHA categories up to the moment of writing this lines:

Backdoor A malware that allows a remote script or program to be executed on the device. However, the executed code can be of any other PHA category. For example, GhostCtrl is a malware that tricks the user to install it and then it connects to a remote server to get commands to execute[3]. It can record audio, video, and run shell commands among other actions.

Spyware An application that transmits user data without his consent to a third party. The transferred data can include SMSs, call logs, photos, contact list, web navigation history, etc. For example, Skygofree is a spyware that starts recording audio depending on the location of the device, for example when a person enters his office[4].

Denial of service An application that participates in a denial-of-service attack. For example, WireX is a botnet created to generate a DDos traffic. It was available on Google Play store. It was detected in August 2017 and took down thanks to a collaboration of several companies like Akamai, Cloudflare, Flashpoint, Google, Oracle Dyn, RiskIQ, and Team Cymru[5].

[2] http://blog.avlyun.com/2014/12/1978/poisoncake-in-the-romenglish-version/

[3] https://blog.trendmicro.com/trendlabs-security-intelligence/android-backdoor-ghostctrl-can-silently-record-your-audio-video-and-more/

[4] https://www.kaspersky.com/blog/skygofree-smart-trojan/20717/

[5] https://blogs.akamai.com/2017/08/the-wirex-botnet-an-example-of-cross-organizational-cooperation.html

Hostile downloader An application that downloads other harmful applications. It may be harmless *per se* but the fact that it allows other malware to be installed makes it malicious.

SMS fraud An application that sends premium SMS messages without the user consent. ExpensiveWall is an example of this category of malware[6]. It was downloaded more than 1 million times from Google Play.

Call fraud An application that charges the user by making premium calls without his consent. An example of this PHA category is MouaBad.p[7].

Phishing An application that pretends to be from a legitimate source in order to steal the user credentials such as credit card credentials and online login information. Roaming Mantis, for example, is a malware that uses DNS hijacking to infect Android smartphones. It redirects users to malicious IP addresses to install trojanized applications[8].

Privilege escalation An app that breaks a security functionality like sandboxing or disables SELinux to gain more privileges, for example to prevent its removal. If it roots the phone, it can be classified in the rooting category.

Ransomware An app that takes control of the device or user data by locking the phone or encrypting the data and asking for a ransom to release it. For example, Simplocker encrypts users multimedia files stored in the SD card and asks a ransom to decrypt them[9].

Rooting A privilege escalation app that roots the device without user consent. This is different from rooting apps that power users download intentionally to root their phones. For example, SpyDealer is a spyware that uses tooting techniques to enable subsequent data theft[10].

[6] https://blog.checkpoint.com/2017/09/14/expensivewall-dangerous-packed-malware-google-play-will-hit-wallet/

[7] https://blog.lookout.com/mouabadp-pocket-dialing-for-profit/

[8] https://securelist.com/roaming-mantis-uses-dns-hijacking-to-infect-android-smartphones/85178/

[9] https://www.zscaler.com/blogs/research/analyzing-android-simplocker-ransomware

[10] https://researchcenter.paloaltonetworks.com/2017/07/unit42-spydealer-android-trojan-spying-40-apps/

Spam An app that sends unwanted commercial ads to user contacts, or uses the device as an email spam relay.

Trojan An app that pretends to be harmless but performs some harmful operations in the background without the user consent. This category of apps can also be classified in another category depending on the harmful actions they do.

3.1.1 Analysis Escaping Techniques

Android malware uses multiple techniques to bypass security tools. It uses code obfuscation to escape static analysis based tools. In addition, it employs several mechanisms to escape runtime vetting solutions. Most of these techniques were initially used to protect author intellectual property, but they have been broadly used by malware authors to evade detection and defeat signature based anti-viruses. Evasion techniques have even been used to automatically generate Android malware that are unrecognizable by most anti-viruses. For instance, G. Meng et al. has proposed a framework that leverages genetic algorithms to generate new malware samples using an initial set of attack and evasion features [44]. They tested the generated samples on 57 off-the-shelf anti-viruses tools and 9 academic solutions, and showed that only 30% of the samples were detected.

Code Obfuscation

Malware uses several code obfuscation methods available from off-the-shelf tools like ProGuard, DexGuard, APK Protect, HoseDex2Jar and Bangcle. For example, ProGuard is included by default in Android SDK[11]. It replaces class, method and variable names with new names like `a.a()`. But it can also use a custom dictionary. DexGuard is a commercial and more advanced and sophisticated version of ProGuard[12]. It replaces class and method names with non ASCII characters and encrypts strings. In contrast to the other tools, Bangcle is an online service that modifies application heavily[13]. It can even encrypt the bytecode and load it dynamically at runtime.

Code obfuscation is one of most used evasion techniques because of its simplicity. Reflection, for instance, is an advanced feature of the Java language that allows to dynamically change the execution flow. It is slower than a simple method invoke, but it is

[11] `https://www.guardsquare.com/en/products/proguard`
[12] `https://www.guardsquare.com/en/products/dexguard`
[13] `https://www.bangcle.com/`

```
1 String var1 = "Y29tcGxldGUuY2xhc3NwYXRoLmFuZC5Gb28="; // Foo
2 Object foo = Class.forName(new String(Base64.decode(var1,
      Base64.DEFAULT))).newInstance();
3
4 String var2 = "aGVsbG8=";       // hello
5 Method m = foo.getClass().getDeclaredMethod(new
      String(Base64.decode(var2, Base64.DEFAULT))), new Class<?>[0]);
6 m.invoke(foo);                  // foo.hello()
```

LISTING 3.1: Code obfuscation using Java reflection

used by malware to protect the sensitive parts of their code in order to make it difficult to detect by static analysis tools. The character string containing class and method names can be obfuscated or encrypted. Listing 3.1 shows an example of object instantiation and method call using reflection. To make matters worse, the class and method names are obfuscated in this case, which increases the difficulty of understanding the code. This can make manual inspection slow and automatic analysis challenging.

Several works were proposed to tame reflection in Android apps. Some approaches operate on the bytecode without the need to run the application in order to find what methods are called by reflection. For instance, Zhang et al. propose Ripple, a static reflection analysis tool for Android that resolves reflective calls to improve data leak detection [45]. Similarly, Li et al. proposed DroidRA, a tool that reduces resolution of reflective calls to a composite constant propagation problem [46]. It operates statically on the Jimple code and does not execute it.

In contrast to the other works, S Rasthofer et al. proposed a tool called Harvester that runs slices of the code in order to extract obfuscated values [20]. The goal of Harvester, therefore, is not to keep the original context of the apps. It merely extracts runtime values that an analyst or a tool can use to eliminate obfuscation and bring runtime information into the static analysis world.

Runtime Analysis Evasion

To escape runtime analysis tools, Android malware uses time and system events to delay the execution of the malicious actions. These techniques are rarely used by benign apps to hide their intents. Therefore, the presence of triggers is an good indication that the app needs further scrutiny [11]. BadNews, for instance, is a malware that waits

for commands from a remote server and performs actions depending on them, such as downloading and installing an APK file or creating a notification with a URL to open [23].

Android malware can also delay the malicious code execution using time bombs. For instance, MobiDash is another malware, an adware that has 3 states, *None*, *Waiting* and *WaitingCompleted* [23]. After installation, the malware enters the *None* state, and changes its state to *Waiting* when certain system events occur like a device reboot. MobiDash then starts waiting for a pre-configured duration that can go up to several days. After, it changes its internal state to *WaitingCompleted* and launches the malicious code that shows unwanted ads whenever the user turns on the screen.

Runtime analysis evasion has already drawn the attention of the research community. For example, in a recent study, M Wong et al. showed that 80% of their 2000 analyzed malware samples have runtime evasion techniques. Consequently, authors propose Tiro, a tool that can automatically detect and reverse runtime based obfuscation techniques. Several other works were proposed to mitigate runtime obfuscation mechanisms. For instance, G. Suarez-Tangil et al. proposed a method to detect malicious code even if it is obfuscated at runtime [47]. They suggest that malware may load code from assets, images, etc. or use an obfuscated string to connect to a URL for example. Authors inject faults in the suspicious parts of the app (images, videos, strings, databases etc.), and if the behavior changes from the original one they conclude that this part is malicious.

3.2 Analysis Approaches

Due to the sensibility of the data that mobile devices carry and the economical danger of security breaches on these devices, several research projects were started and hundreds of articles were published in the last decade about Android security. In this section, we explain the most relevant Android application security approaches proposed in the literature that are connected to our contributions. These approaches fall into three categories: protection, detection and characterization approaches.

3.2.1 Protection Approaches

Enforce the security measures in the first place on Android platform is a good idea to fight against malicious software. Thus, numerous works were proposed about this

subject [48]–[53]. For instance, G. Tuncay et al. try to solve the problem caused by custom permissions because Android treats them as it does with system permissions, even though they are from different trust levels [54]. Authors propose a system called Cusper that handles custom permissions securely. Similarly, to prevent apps from colluding to achieve inter-app malicious behaviors, S. Rasthofer et al. proposed DroidForce, a tool that checks inter-app data flows and operates in the app space without rooting the device [55]. It injects security enforcement into applications using Soot, where in every sensitive operation, a request is sent to the Policy Decision Point (PDP). Consequently, if the permission is denied, the application skips to the next instruction. Android is an agile system where applications can access each other's components, but it gives rise to security concerns. In [56], C. Yagemann et al. proposed a tool that can intercepts intents in Android system and handles them with a user app. Thus, it can block and allow intents through configurable policies.

Indeed, these techniques help to strengthen Android security mechanisms. However, they do not address permission misuse and vulnerability exploitation. Thus, detection approaches are needed to limit malware proliferation.

3.2.2 Detection Approaches

Most of the state-of-the-art works on Android security fall in this category. Malware detection is important because of the need to protect users from the huge number of Android malware uploaded to Internet every day. Detection approaches can be split into two types depending on the used analysis technique: static or dynamic.

Static Analysis Based Detection Approaches

Static analysis based tools can be very practical because of the relatively quicker processing time. Thus, several approaches try to detect Android malware using static features like code structure, permissions, meta-information and package properties [57]–[63].

Sensitive data leaks have causes significant privacy and security concern. V Avdienko et al. proposed a tool called MUDFLOW that detects malware basing on wrong handling of sensitive information [64]. Similarly, S. Arzt et al. proposed FlowDroid, a tool that looks for data leaks by following information flows from sinks, such as socket, backward to sources, like where a method writes to a socket [6]. Flowdroid is not path sensitive, and app components (Activity, Service, Content provider, Broadcast receiver)

are called in undefined (asynchronous) order from a dummy main method. In addition, implicit data flows are ignored; those that are related to control flow dependencies. Most Android data flow analysis systems use a list of sources and sinks to manage their data security policy. Consequently, a lack of information about sensitive sources or sinks may lead to a security breach. For instance, if Android adds a new way to send information, it should be integrated in the sinks list. Otherwise, malware can use it to leak data. FlowDroid is configured to use Susi to get the list of sources and sinks for its tainting process. Susi is a tool that uses supervised machine learning algorithm to classify methods to sources, sinks and neither [65].

Other static app features have also been used to detect malware. For instance, DROIT analyzes meta-information on the market store and the Android manifest, and scans them on different anti-virus engines (learning process) that classify apps as benign and malicious [66]. It uses natural language processing techniques to classify apps by different machine leaning algorithm like: Random forest, k-Nearest neighbors, Decision trees, AdaBoost, Bagging, and Naïve bayes. The code structure has also been used to classify apps. RiskRanker, for instance, divides potential risks that an app may constitute into three categories: high-, medium-, and low-risk, depending on code patterns like socket writing and message format [67]. Another tool that uses code structure to classify malware into families is Dendroid [68]. It represents the code as a grammar and uses Vector Space Model (VSM) to represent the samples. It calculates similarities between apps using those vectors, and clusters them using a dendogram. Similarly, Drebin uses machine learning on static features such as permissions, API calls and network addresses to detect malware [57]. And DroidSieve collects information on API calls, code structure and permissions to detect families of malware using the machine learning algorithm Extra Trees [69].

Dynamic Analysis Based Detection Approaches

Dynamic analysis tools have the advantage of data that are only available at runtime. In addition, they help to dynamically analyze malware behavior if the code is obfuscated. Thus, they are effective in detecting some advanced malicious behaviors [13], [58], [70]–[72]. For instance, S. Dash et al. proposed a tool called DroidScribe that classifies malware into families by observing exclusively the runtime behavior of the sample [73]. ProcHarvester is another runtime analysis tool that detects *procfs* side-channel information leaks by launching triggering events and using machine learning to detect leaks [74].

Dynamic analysis is very helpful to analyze malware with obfuscated code, but it is generally slower compared with the fast static analysis. This explains why there are less dynamic analysis based security tools compared with their static analysis counterparts. However, they can be used to characterize malware, which we will explain in the next section.

3.2.3 Characterization Approaches

Characterizing malware and its behavior is an intrinsic part of understanding malware effectively. This is particularly useful where the analyzed application is not straightforward to detect as malware. In this case, understanding the app's characteristics could help for better classification. In [7], S. Poeplau et al. study this issue in benign and malicious apps. They discovered that many Google Play apps are vulnerable to malicious code injection through dynamic loading. The authors propose to add mandatory checks in the operating system to vet the externally loaded code before execution. Besides, malware authors tend to inject their code in benign apps to facilitate the malware creation process and to disguise it. In [75], Li Li et al. locate the packages responsible for malicious behaviors in piggypacked (repackaged) apps. They automatically rank potentially malicious packages by identifying the original app (carrier), the injected malicious code (rider), and switching code (hook). This characterization is useful to understand how malware authors repackage their code in benign app. To achieve this, the authors made a directed graph of dependency (calls between packages) where the wight represent the number of calls.

Other existing approaches base their characterization on runtime information. For example, Yan et al. propose DroidScope, a virtual machine introspection based analysis tool that uses a virtualized OS and instruments the Dalvik VM in order to track taints applied to the app [76]. It is built on top of QEMU [77] and aims at reconstructing the semantic of high level information about the app. Another characterization tool that uses virtual machine introspection is CopperDroid [10]. It aims at better understanding of the malware behavior at runtime by adding a layer above Qemu to intercept syscalls and construct Android specific objects. CopperDroid reconstructs the app behavior outside of the operating system. It looks for suspicious actions like accessing personal info, executing external applications, altering the file system, and making calls. Furthermore, new approaches were proposed to provides a comprehensive view of malwares behaviors on the new Android runtime (ART). For instance, Malton is an on-device

non-invasive analysis platform for ART [78]. It performs multi-layer monitoring and information flow tracking at runtime.

Due to the importance of better understanding malware and its behavior, our contributions are focused on the characterization part. Since control flow graphs help to understand malware, the next section will be on how to build better CFGs for Android apps.

3.3 Implicit Inter-procedural Calls

Several recent works try to automatically characterize malicious behavior [12], [19], [28], [29]. They leverage a combination of static and dynamic analyses. A first static analysis of the code identifies the most suspicious locations in the code and then a particular run of the application targets the execution of the code previously identified as suspicious.

For instance, FlowDroid [6] which detects data leaks by following information flows, uses a dummy main method to simulate the Android life cycle and invoke all the application entry points. In Flowdroid, only method calls can be the origin of a taint. Thus, Flowdroid relies on interprocedural control flow graphs to properly propagate taints. Similarly, ConDroid launches the suspicious code in apps to reveal any malicious activities [28]. It searches execution paths from entry points, such as life-cycle methods and input events, to the suspicious code locations. Then, it performs an adaptive concolic execution. It instruments the app and modifies variable values, in order to observe the application's behavior. ConDroid also uses the application's interprocedural control flow graph to perform the concolic execution. GroddDroid is another tool that automatically triggers and monitors the suspicious code in Android apps [19]. It locates the code considered as suspicious, which may be protected or hidden (ciphered, encoded, obfuscated or dynamically loaded), or when it has a call to a sensitive API method such as sending an SMS [79]. GroddDroid also exhibits execution paths from the entry points to the suspicious code. Then, it instruments the app by forcing the necessary branches in the execution path to reach the malicious code when the malware is executed. Furthermore, researchers have discovered that malware targeting desktop systems keep their high-level properties of code, such as the call graphs [80], [81]. In [82] for instance, H. Gascon et al. use application's call graph to detect new variations of known malware by machine learning.

These approaches strongly rely on a good computation of global control flow graphs that represent all execution paths in the program [15]. Such CFGs are useful only when they contain the necessary execution paths that lead towards suspicious code locations. Unfortunately, the CFG construction methods that these tools use do not take into consideration all types of execution paths. They merely analyze the application code, which induces missing paths that pass though the Android framework.

3.3.1 What Are Implicit Calls?

Android application code is compiled into Dalvik bytecode, as we mentioned earlier. Dalvik bytecode can be translated into Jimple intermediate representation [34] by Soot [35]. This makes it easy to build a CFG for each method independently at the granularity of a Jimple statement, where an oriented edge between a node $\mathscr{A}$ and a node $\mathscr{B}$ indicates that statement $\mathscr{B}$ is to be executed immediately after statement $\mathscr{A}$.

Building method CFGs constitutes an important step towards accurate static analysis. However, we are interested in the *global* or *interprocedural* CFG that contains all execution scenarios for a given application, because it shows how the malicious code is protected, and how it can be launched. The global graph is constructed by connecting all the method CFGs, i.e., by adding edges representing interprocedural calls. There exist two types of interprocedural calls: *explicit* and *implicit*.

Explicit Call: A method `a()` *explicitly* calls a method `b()` if the code of `a()` contains a call (an `invoke` statement) of `b()`. For example, in Listing 3.2 line 11, statement `run(content)` is an explicit call of method `MailTask.run(String)`. In this case, We can build an edge from the node representing the invoke statement `run(content)` in method `doInBackground()` towards the node containing the first statement in the CFG of method `run(String)`. This is represented by a solid arrow on the right part of Figure 3.1

Implicit Call: A method `a()` *implicitly* calls a method `b()` when the following conditions hold:

1. `a()` contains a call (an `invoke` statement) of a method `c()` which is defined in the Android framework.
2. `c()` invokes `b()` either directly or through a sequence of method calls in the framework that ends by an invocation of `b()`.

For example, method `doInBackground()` in Listing 3.2 is implemented in the application code but invoked by the Android framework. This method does not have an incoming control flow edge starting from the application itself, and that is why such methods are called callbacks. If a malicious code is located in a method which is implicitly called, it will be considered as unreachable by most existing static analyzers since they do not take into account the Android framework.

Listing 3.2 shows an example of code extracted from a real-world spyware[14] that sends sensitive information such as the device ID and the contact list to a remote server. The entry point of this malware sample is `ClientActivity.onCreate()`. The most interesting part of code is mainly the malicious statements `HttpSend.postData(url, localArrayList)` appearing in method `run(String)`. This statement leaks sensitive information previously retrieved by calling `context.getSystemService("phone").getDeviceId()`. The main goal of dynamic analysis is thus to observe this application executing the suspicious method `run(String)`. Method `doInBackground()` is *implicitly* called when `MailTask.execute()` is executed.

Obviously, the CFG of this code which is depicted in Figure 3.1 could be incomplete if we do not take into consideration the implicit call going from `MailTask.execute(new Integer[0])` (line 5) to `MailTask.doInBackground()` (line 9). Furthermore, running directly `MailTask.execute()` by instrumenting the application without finding a complete path from an entry point is meaningless since the suspicious method will be isolated from its context and could not have access to objects built in `ClientActivity.onCreate()`. A standalone analysis of the app code could not reveal the existence of such a call. Consequently, we have to analyze additional code outside the application, namely the Android framework to determine implicit calls and build a reliable interprocedural CFG.

3.3.2 Implicit Calls in the Literature

Being able to take into account implicit calls appears to be a key point for improving recent works on static analysis of Android malware. For instance, Flowdroid [6] achieves static taint-analysis of Android applications and relies on CFGs which are computed from various sources, including layout XML files, executable code and the manifest file. This work should benefit from our computation of a global CFG that takes the framework into account. In the same way, Lillack et al. use taint analysis to know

[14] SHA-256: 45d21e32698d1536a73e42c1e5131c29ca94b9d9d1bd5c744bd74ffc2af6853e

```
1  public class ClientActivity extends Activity {
2     protected void onCreate(Bundle bundle) {
3        /* ... */
4        MailTask mt = new MailTask("", ((Context) this))
5        mt.execute(new Integer[0]);
6     }
7  }
8  public class MailTask extends AsyncTask {
9     protected String doInBackground (Integer... args) {
10       /* ... */
11       run(content);
12       return "doInBackground:" + this.content;
13    }
14    public void run(String arg) {
15       /* ... */
16       String str2 = ((TelephonyManager)
             context.getSystemService("phone")). getDeviceId();
17       ArrayList localArrayList = new ArrayList();
18       localArrayList.add(new BasicNameValuePair("imei", str2));
19       localArrayList.add(new BasicNameValuePair ("count", Integer.
             toString(i)));
20       localArrayList.add(new BasicNameValuePair("notebook", "Number:" +
             i + "\r\n" + str1));
21       String url = "######.com/MailTask.php";
22       HttpSend.postData(url, localArrayList);
23    }
24 }
```

LISTING 3.2: Implicit call in a real-world malware

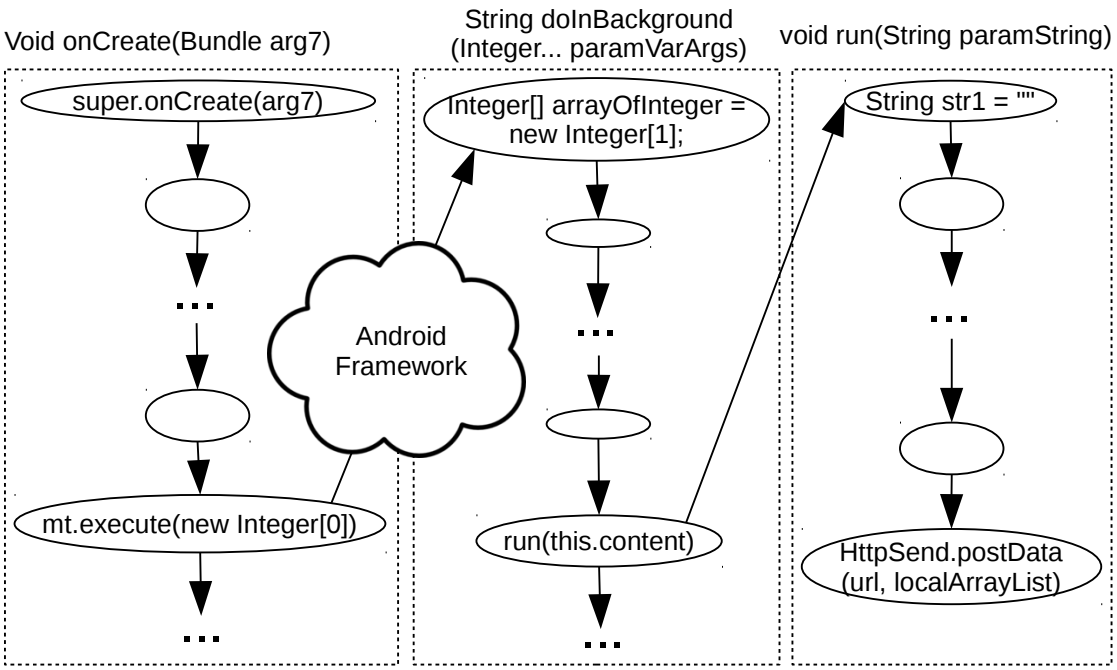

FIGURE 3.1: Global control flow graph with implicit calls

which parts of an Android application are influenced by the platform's configuration, e.g. when Bluetooth is activated [83] . Klieber et al. [84] rely on FlowDroid for intra-component taint analysis, and on Epicc [85] for inter-component analysis. This work handles calls that occur when an activity calls another one to propagate the taint. Nevertheless, authors do not propose a solution for other types of implicit calls, which leads to imprecise results. Graa et al. aimed at to make FlowDroid handle the control flows that leak information implicitly [86]. They mainly focused on implicit flows that occur due to conditional branches. Nevertheless, they also did not take into consideration implicit calls generated by the Android framework.

Some approaches have made attempts to handle some callbacks. Wu et al. [87] build callback graphs for synchronous callbacks, like for classes `AsyncTask` and `Handler`, in addition to application components, namely *Activity*, *Service*, *Broadcast Receiver*, and *Content Provider*. Authors focus only on main classes and methods, and neglect other callbacks that may be called by the framework. In [88], authors use lifecyle callbacks of Android applications to build a model of the application and then detect malicious behaviors. This approach focuses only on lifecyle callbacks and does not handle other types of implicit calls. Furthermore, in [89], authors perform context-sensitive analysis of callbacks and statically generate GUI models for Android. However, Android framework defines thousands of callbacks, but the authors of this paper focus *only* on the lifecycle and interactions of user event-driven components.

None of these works is able to handle most of the implicit calls due to the Android framework itself. Hence, Android malware can easily leverage implicit calls to escape these approaches that lack for precision. In Chapter 4, we present our approach to cope with implicit calls in the Android.

3.4 Triggering Approaches

Purely static analysis tools have become less effective because recent malware is using code obfuscation techniques like reflection and string encryption. Dynamic analysis is needed in this situation to complete the vetting process. Thus, approaches of the last ten years try to reach the point in the code where the code seems to be obfuscated or dynamically loaded, in order to observe it at runtime [10], [12]–[14]. Techniques that rely on monitoring the runtime behavior face another type of challenges. Android malware may not expose its real behavior except under narrow conditions [11], [16]. The malicious code can stay dormant for a specific duration before executing the payload. In addition,

several system events can be used to differ launching the malicious code. For instance, the malicious code can wait for an SMS reception, a reboot of the phone, or any other event to launch the malicious actions. A manual inspection can be useful to reverse a sample and identify the experimental conditions to get a successful observation at execution time. This reversing effort cannot be applied for the thousands of applications uploaded every day to the different online application stores. Consequently, to observe the malicious code executing, it should be *triggered* in the first place.

Some previous works have already worked on the specific problem of triggering Android malware behaviors. Fratantonio et al. have proposed a new static approach that helps to detect the triggering conditions that have great chances to protect the malware payload [11]. Nevertheless, authors do not evaluate the performance of successful triggering at execution. Android malware are known to use different techniques to hide their malicious code. A tool called HsoMiner has been proposed by Pan X. et al. to detect unknown hidden sensitive operations basing on code hiding techniques [16]. Malware tends to used triggering conditions that are unrelated to the malicious code, like checking the date then sending an SMS. Malware tries to detect if the device is an emulator before launching the malicious actions to avoid analysis by security tools. HsoMiner uses these properties to characterize Android malware, even though few benign developers do use code hiding techniques to secure their apps and protect their intellectual properties.

Depending on the operational method, recent triggering works can be classified into two categories. The first category focus on triggering the targeted code from outside the application. They solely stress the user interface or apply changes to the underlying environment by injecting framework events and/or intercepting API calls and modifying them. For instance, in [90], A. Salem proposes to detect and stimulate Android repackages malware using active learning. He uses a simple method to stimulate apps which emulates the user-app interaction by starting the main activity, retrieving its UI elements and interacting with them randomly. Some tools aim at launching the malicious code by instrumenting the graphical user interface. In [17], S. Hao et al. presents PUMAscipt, a scripting language that automates Monkey[15] execution and collects runtime information or triggers changes in the environment. It exposes a programmable interface with a script language to interact with the app and customize the dynamic analysis. CopperDroid also stimulates the app, but by sending events like location updates and SMS

[15] A program that runs on the emulator or the device and generates pseudo-random user events such as clicks and gestures

reception [10]. It analyzes the system calls generated by the application in order to capture the objects that are involved in these events. CopperDroid uses a classical static extraction of the possible events from the Manifest. A more recent approach, called Fuzzdroid [18], has obtained more precise experiments. It builds iteratively the execution environment that helps the path to be steered towards the malicious code. Similarly, in [12], authors propose IntelliDroid, a tool that aims at finding the required inputs and their occurrence order to trigger the suspicious code at execution time. These inputs are injected using low-level device-framework events in order to preserve the context of the application's execution. IntelliDroid computes the dependencies between the used variables that compose the conditions on the execution path. It also collects the possible definition of the required objects in the code and their associated value at runtime.

These approaches fail to trigger some complex logic bombs where reversing a condition is not straightforward, for example, when a triggering condition compares a hashed value of an input value against a stored value. Obviously, reversing a hash function to guess the right input to provide is a hard problem.

The second category of triggering approaches focuses on triggering the targeted code with little attention paid to the application context. For example, A. Abraham et al. proposed GroddDroid, an analysis framework that triggers suspicious code mixing UI stimulation and bytecode alteration [19]. However, GroddDroid merely replaces `if` statements with `goto`s to force the control flow to take a certain branch. Applications analyzed by this method generally know a lot of crashes due to brute manner of control flow steering. Harvester is an extreme example of tools that do not pay attention to the application's context. It extracts and executes slices of the code [20]. However, the main purpose of Harvester is to extract obfuscated runtime values.

These techniques can be useful for testing purpose, but they can expose unrealistic behaviors by removing part of the code that are necessary to understand the application's overall behavior. This partial view of the app's behavior can mislead the analysis tool and extract partial or even wrong information.

Works presented in this section either focus on launching the targeted code without preserving the original context or preserving the application context but fail to trigger non simple conditions. In Chapter 5, we present our approach to trigger the suspicious code in Android applications while preserving most of the application's original context.

3.5 Conclusion

In this chapter we explored the most relevant works on Android application's security. We focused especially on approaches that try to enhance the characterization of Android malware. Then, we also explored the state-of-the-art works that aim at solving the problem of implicit calls and incomplete control flow graphs. We will present our contribution to the subject of CFG and execution paths in Chapter 4. I addition, we presented recent works that aim at triggering the suspicious code in Android applications. We will also present our contribution to suspicious code triggering and malware characterization in Chapter 5.

Chapter 4

GPFinder

Several Android malware detection approaches rely on the computation of application global control flow graphs (CFGs) that represent all execution paths in the program in order to automatically characterize the malicious behavior [12], [19], [28], [29]. They perform a static analysis of the application in order to locate the suspicious code locations. Then, they launch a specific execution of the app in order to trigger the targeted code previously identified as suspicious. Such CFGs are useful only when they contain the necessary execution paths towards the suspicious code. Unfortunately, these approaches do not take into consideration all types of execution paths because they only analyze the application code, which leads to miss paths that pass though the Android framework.

In this chapter, we present our approach on how to automatically exhibit execution paths towards suspicious locations in the code by computing global CFGs that include edges representing explicit and implicit interprocedural calls. We implemented our approach in a tool called *GPFinder* (GroddDroid Path Finder) as the main practical outcome of this work. GPFinder helps security analysts retrieve execution paths that may trigger the malicious code, even when they pass through Android framework's callbacks. It also gives key information about the analyzed application in order to understand how the suspicious code was injected into the application.

To validate our approach, we use GPFinder to study a collection of 14,224 malware samples, and we show that including implicit calls to build CFGs improves the analysis. We evaluate that 72.69% of the samples have at least one suspicious code location which is only reachable through implicit calls. Furthermore, we investigate the common structures of Android malware, we highlight their favorite entry points and how they use implicit calls.

This chapter is structured as follows. Section 4.1 explains the design of GPFinder,

and how it operates, and it discusses how the Android framework can be mined to complete applications' CFGs with implicit interprocedural calls. Next, Section 4.2 details how GPFinder benefit from these CFGs to study the inner structure of an Android malware set. Then, Section 4.3 shows the important findings of this experiment. In Section 4.4, we analyze a dataset of benign applications and compare the findings with those of the malicious applications. We discuss the results in Section 4.5 and conclude the chapter in Section 4.6.

4.1 Approach

We showed in Section 3.3 that many security tools rely on global CFGs to launch suspicious codes found in Android applications. These CFGs should contain all the necessary execution paths towards the suspicious code in order to reach them. This means, in addition to method CFGs and explicit interprocedural call, global CFGs should also include implicit calls, i.e., those that are generated by methods implemented by the application and invoked by the Android framework (*callback methods*). For this purpose, we need to analyze the Android framework, because these calls pass through its API. Analyzing solely the application code could not reveal the existence of such calls.

Implicit calls pass through the Android framework using pairs of methods *registration-callback*. A callback method is a method implemented in the apps code, it overrides a framework method and can be called by the framework. A registration is a method implemented in framework space. It communicates the availability of a callback to the framework. It calls the callback method directly or through a sequence of method calls inside the framework space. Listing 4.1 shows a pair of registration-callback methods (`sort(List,Comparator) - compare(Object,Object)`) that generate an implicit call. In this example, `process`) does not call directly `compare`, but it calls the registration method `sort` which is implemented in the Android framework. Method `compare` is called, in this case, by the framework.

In order to find if there exist an implicit call between two methods in the application code, we need to analyze the Android framework. We can manually figure out some implicit calls, but not all of them. We need to automatize the process of analyzing the Android framework to get the list of all possible implicit calls.

Cao et al. were already concerned about the lack of these callbacks in global CFGs [91]. They have implemented their approach in a tool called EdgeMiner. It performed a static analysis on the 24,089 classes of the Android framework and extracted

```
1  void process(){
2     List<Integer> list = new ArrayList<Integer>();
3     ...
4     MalComp mal = new MalComp();
5     Collections.sort(list, mal);
6  }
7  ...
8  class MalComp implements Comparator {
9     int compare(Object arg0, Object arg1) {
10       /*- Do bad stuff */
11       return 0;
12    }
13 }
```

LISTING 4.1: Registration and callback methods

a list of 5,125,472 *registration-callback* pairs responsible for implicit control flow calls. These summaries are under the form of `registration#callback#position`, where the `registration` and the `callback` are the involved methods, and the integer `position` denotes the place of the `registration`'s argument responsible for calling the `callback` method.

In this dissertation, we use EdgeMiner to get the list of all possible implicit calls that pass through the Android framework. We combine the analysis of the Dalvik class hierarchy with the EdgeMiner generated summaries in order to compute a global CFG with implicit edges. With a global CFG computed by our tool GPFinder, we intend to find all execution paths leading towards targeted methods in the application bytecode, especially the suspicious ones. The overall architecture of GPFinder is depicted in Figure 4.1. It shows how GPFinder takes in input an Android application and the framework summaries and generates the global control flow graph and other useful information about the analyzed application.

For instance, the following generated EdgeMiner's summary shows an implicit calls between the registration method (`sort(List,Comparator)`) and the callback method (`compare(Object,Object)`). The link between these two methods is the second parameter `Comparator` of the registration method which is of the same type as the callback class. This information is given by the `position 2` indicated in the following EdgeMiner summary.

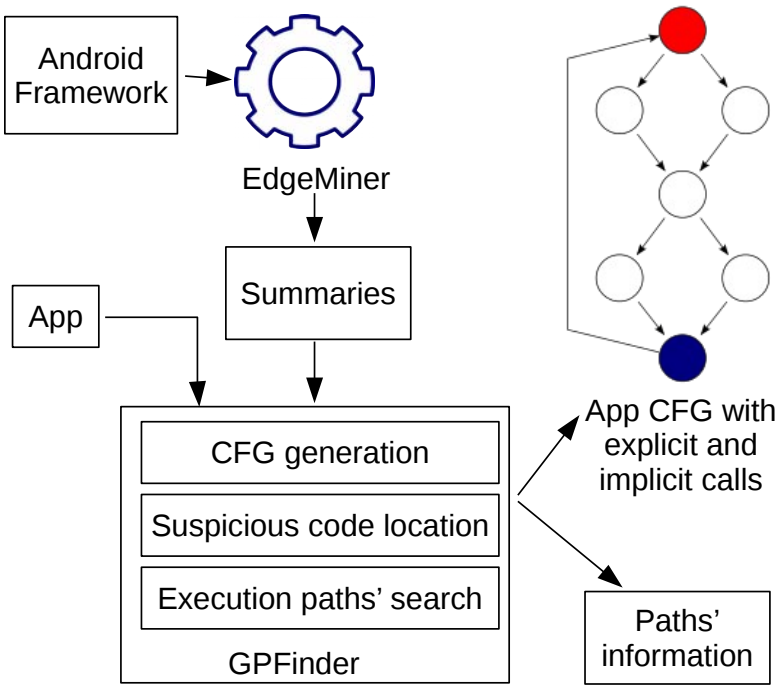

FIGURE 4.1: GPFinder's architecture

```
java.util.Collections: void sort(java.util.List,java.util.Comparator)
# java.util.Comparator: int compare(java.lang.Object,java.lang.Object)
# 2
```

LISTING 4.2: EdgeMiner rule

Note that if the `position` equals zero, it means that the defining class of the callback method should be the same as the defining class of the registration method.

4.1.1 Control Flow Graphs Generation

Global CFGs of Android applications need to include the necessary edges in order to be useful for security analysis tools. In other words, global CFGs need to have: Method CFGs, explicit interprocedural calls, and implicit interprocedural calls. GPFinder first generates one CFG per application method. Then, it uses the Spark [40] module of Soot to generate explicit interprocedural calls and connect method CFGs. Spark is a flow-insensitive and implements a points-to analysis using the subset-based [41] and equality-based [42] approaches.

Next, GPFinder finds implicit interprocedural calls between method and uses them to add more edges into the application global control flow graph. GPFinder finds implicit interprocedural calls as follows:

> For each pair `(invoke(b()), a())` where `b()` is a framework method and `a()` is a method overridden in the application code, GPFinder add an edge from node `invoke(b())` to node `a()`
>
> $iff.$
>
> There is a rule `registration#callback#position` in EdgeMiner summaries where `b()` equals or overrides `registration` and `a()` overrides `callback`.

A method `x()` overrides a `callback` or a `registration` method when the following matching conditions hold:

(a) **Name:** The overriding method in the application code has the same name as in the EdgeMiner rule.

(b) **Defining class:** The defining class of `x()` is a subclass of the one defining the corresponding `callback` or `registration` method.

(c) **Return type:** The type returned by `x()` is a subtype of the one returned by the corresponding `callback` or `registration` method.

(d) **Arguments:** Each argument of `x()` is the same or a subtype of the corresponding argument of the `callback` / `registration` method.

(e) **Position:** If the `position` $p = 0$, the `callback` class must be the same or a subtype of the registration class. If $p > 0$, the `callback` class must be the same as the p^{th} argument of the registration method.

Matching application and framework methods with their EdgeMiner counterparts is not straightforward because of the inheritance mechanism of the Java language. A class can implement different interfaces and override a class. The overridden class may also implement different interfaces and override another class, etc. This recursive inheritance relation that a Java class might have makes matching a method with the registration or callback method of an EdgeMiner rule challenging. For instance, in order to match the registration and callback methods shown in Listing 4.1 with their counterparts in the EdgeMiner rule shown in Listing 4.2, we need to ensure that:

1. The defining class `Collections` of the registration method is the same in the application code as well as in the EdgeMiner rule (Condition (b)).
2. The `MalComp` class is indeed a subtype of the `Comparator` class of the callback method (Condition (b)).
3. The returned type `void` of the registration method is the same in the application code and in the EdgeMiner rule (Condition (c)).
4. The returned type `int` of the callback method is the same in the application code and in the EdgeMiner rule (Condition (c)).
5. The registration method in the application code `sort` has identical parameter types (`List, Comparator`) as in the EdgeMiner rule (Condition (d)).
6. The callback method in the application code `compare` has the same parameter types (`Object, Object`) as is the EdgeMiner rule (Condition (d)).
7. The class of the registration method in the application code `MalComp` is a subtype of the second parameter of the registration method in the EdgeMiner rule `(Comparator)` (Condition (e)).

4.1.2 Suspicious Code Location

GPFinder automatically locates the suspicious code in the application's bytecode. The manner of locating suspicious code is borrowed from GroddDroid which uses a heuristic

function to score the statements of the application [19]. Every method of the application gets an initial risk score of 0, and the more it uses sensitive APIs the more its score increases. The sensitive API methods are divided into categories related to networking, telephony, cryptography, binary code execution, SMS, and dynamic code loading. Nevertheless, it is possible to remove or add any method, class or category to this list. GPFinder sets a risk score for each category and computes the total risk of each application method by adding the risk scores of the API methods it calls. Methods with non-zero scores are considered as suspicious and become subject of further analysis. We slightly modified the scoring function presented in GroddDroid to get better results. These are the adopted scores of the suspicious API categories:

- Binary code execution: 6
- Dynamic code loading: 8
- Cryptography: 3
- Networking: 3
- Telephony: 8
- SMS: 50

This method of locating the suspicious code is perfectible. In this dissertation, we used the heuristic function presented in GroddDroid. Indeed, the manner suspicious code is tagged is important for the results, but this dissertation does not intent to contribute on this subject. In the last months, an intern was recruited in our research team to test and improve this heuristic function. Nevertheless, it was after the publication of the GPFinder's work.

4.1.3 Execution Paths' Search

After locating the suspicious code, GPFinder tries to find a path per suspicious location. It takes into consideration the global CFG of the application and the entry points that can be used to launch the application. For example, it can be launched using `Activity.oncreate` to start a user interface when the user clicks the application's icon. Applications also launch with `BroadcastReceiver.onReceive` which can be configured to start when a system events occurs like an SMS reception, a competed

boot of the phone, or when the user unlocks the screen. We will return to detail Android application entry points in Section 4.3.2.

GPFinder details the sequence of method calls, and points out how many conditions protect the malicious code. It uses a breadth-first search algorithm to find execution paths.

4.1.4 GPFinder's Output

GPFinder gives valuable information within a relatively short analysis time for the security experts. It automatically locates the most suspicious code, computes all execution paths towards these suspicious sites and gives insights on how the malware is protected by potential triggering conditions. GPFinder generates for each analyzed application:

- A global CFG that include method CFGs and interprocedural edges representing explicit and explicit calls.
- Scored nodes on the CFG representing suspicious code statements.
- Execution paths that lead to the suspicious code locations inside the application.

For example, we analyzed a spyware that sends sensitive information like the device ID and the contact list to a remote server[1]. Table 4.1 presents a part of the GPFinder's output generated after analyzing this sample. The analysis took 13.6 seconds. GPFinder found 24 instructions related to the telephony API, 92 instructions related to the networking API, and 11 instructions related to the SMS API. This constitutes a total of 127 suspicious instructions distributed in 13 suspicious methods.

GPFinder exhibited 22 execution paths in the global CFG, starting from entry points and leading towards methods considered as suspicious. GPFinder finds multiple execution paths for some methods. This helps to choose the best path if we want to execute the code. These execution paths contains a total of 14 implicit edges, presented in Table 4.1 by the arrow $\rightsquigarrow$. For each execution path, GPFinder points out the implicit calls, the type and number of suspicious instructions, and how many conditions protect the suspicious code.

Using the output and the global CFG generated by GPFinder, we can get a better view of the malware's structure. For instance, this particular piece of malware listens to system events to launch the `onReceive` method unbeknownst to the user. Then, it

[1] SHA-256: 45d21e32698d1536a73e42c1e5131c29ca94b9d9d1bd5c744bd74ffc2af6853e

collects sensitive information about him, and creates a new thread using the `AsyncTask` class to send this data to the attacker via SMS (Path 1 in Table 4.1).

4.2 Experiment

In order to measure the use implicit calls by Android malware and to better understand the structure of Android malware, we performed an analysis that takes as input a collection of 14,224 detected malware samples randomly chosen from a database provided by *koodous.com*. On this malware set, we exhibit all possible execution paths starting form entry points and leading to malicious code locations. The malicious code is here automatically located by a heuristic detailed hereafter, which means that the targeted code is suspicious. This experiment gives an overview of the malware features such as favorite entry points, most frequent malicious code types, the average number of execution paths leading to malicious code locations, the average number of triggering conditions protecting the malicious code from dynamic analysis, and the average number of implicit calls protecting the malicious code from static analysis.

This experiment was launched on an Intel® Xeon® E5-2630 v3 server, with 8 cores, 32G of RAM and Ubuntu as operating system. The global CFG computation takes an average time of 94.23 seconds per sample of an average APK size of 190 kB. The overall experiment took around 2 weeks, and the application were analyzed one after another.

4.3 Findings

Analyzing this big malware dataset gave us valuable information and insights on how Android malware is built and how it operates. In this section, we explore the most important findings.

4.3.1 Suspicious Code Nature

This experiment showed some malware properties that can characterize their overall usage of the Android API. In the whole malware set, we found 159,053 suspicious methods, which correspond to 4.5% of the total methods in the dataset. This means an average of 11.18 suspicious methods per application. Figure 4.2 depicts the ratio of APKs (in orange) found in the malware collection that have a positive risk score divided

Apk: 45d21e32698d1536a73e42c1e5131c29ca94b9d9d1bd5c744bd74ffc2af6853e.apk
Size: 50,112 bytes
Suspicious: true
CFG info: methods: 86, nodes: 1,891, edges: 2,322
13 suspicious method(s) for a total of **127 suspicious instructions** ('telephony': 24, 'network': 92, 'sms': 11)
Analysis time 13.6 sec

14 Implicit transitions in the executions paths towards suspicious methods

• `com.duoji.app.mian.MyReceiver: void onReceive(Context,Intent)`
⇝ `com.duoji.app.mian.Shell: Object doInBackground(Object[])`
• `com.duoji.app.mian.ClientActivity: void onCreate(Bundle)`
⇝ `com.duoji.app.mian.MailTask: Object doInBackground(Object[])`
• `com.duoji.app.mian.Shell: void run(String)`
⇝ `com.duoji.app.mian.SendMessageasync: Object doInBackground(Object[])`
• `com.duoji.app.mian.App: void chkIsFirstRun(Context)`
⇝ `com.duoji.app.mian.HttpSend: Object doInBackground(Object[])`
• Other implicit calls ...

Details of 22 execution paths towards suspicious methods

Path 1 with 8 conditions towards a suspicious method of type(s) {'sms': 2}

`com.duoji.app.mian.MyReceiver: void onReceive(Context,Intent)` *(Entry point)*
⇝ `com.duoji.app.mian.Shell: Object doInBackground(Object[])`
→ `com.duoji.app.mian.Shell: String doInBackground(Integer[])`
→ `com.duoji.app.mian.Shell: void run(String)`
⇝ `com.duoji.app.mian.SendMessageasync: Object doInBackground(Object[])`
→ `com.duoji.app.mian.SendMessageasync: String doInBackground(Integer[])`
→ `com.duoji.app.mian.SendMessageasync: void sendMsg(SmsManager,String, String)` *(Suspicious)*

Path 2 with 1 condition towards a suspicious method of type(s) {'telephony': 2, 'network': 9}

`com.duoji.app.mian.ClientActivity: void onCreate(Bundle)` *(Entry point)*
⇝ `com.duoji.app.mian.MailTask: Object doInBackground(Object[])`
→ `com.duoji.app.mian.MailTask: String doInBackground(Integer[])`
→ `com.duoji.app.mian.MailTask: void run(String)` *(Suspicious)*

Other paths ...

→ = Explicit call ⇝ = Implicit call

TABLE 4.1: Partial output of GPFinder

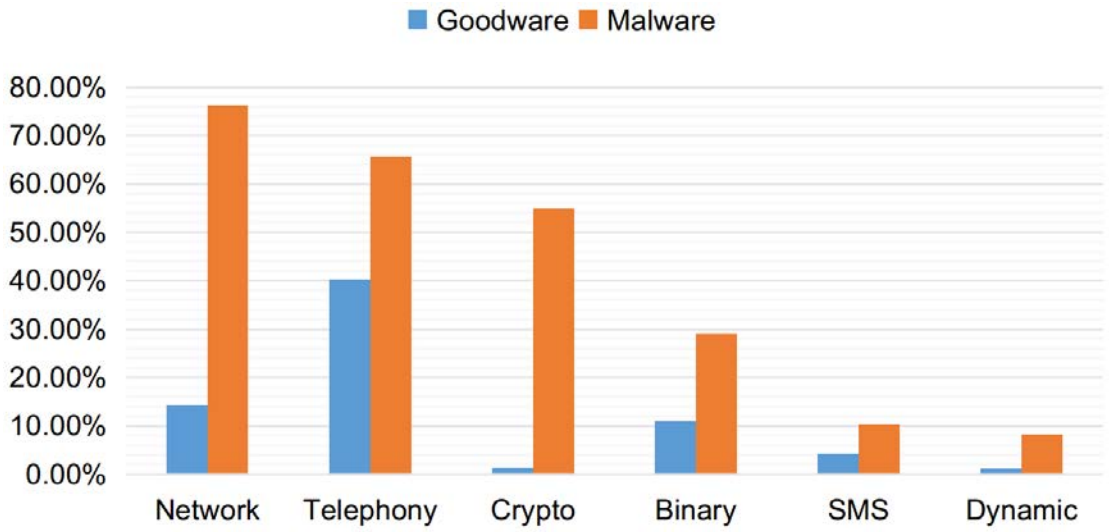

FIGURE 4.2: Suspicious APIs use

by categories of suspicious code (cf. Section 4.1.2 for suspicious code categories). The results show clearly how malware is using the telephony API mainly to collect information about the user like the IMEI and the phone number and send them to remote servers.

4.3.2 Entry Points Types

Android applications can be launched by a number of events, such as when the app launcher is pressed, an Intent is received, etc. Consequently, an application does not have only one entry point but a set of entry points like lifecycle callbacks (`onX()` methods). There exist mainly seven entry lifecycle callbacks belonging to three main categories: Callbacks allowing to create, start or resume an `Activity`, those enabling to create, start or bind a `Service`, and lastly a callback (`BroadcastReceiver: void onReceive`) that wakes up the application when it is notified by a system event.

Among these entry points, we evaluate which ones are the most used to reach the suspicious code. The results of this experiment which are depicted in Figure 4.3 (in orange) reveal that malware prefer `BoadcastReceiver.onReceive(Context, Intent)` and `Activity.onCreate(Bundle)` over other entry points. The use of the latter is common as it enables to launch applications using their launcher icon. Nevertheless, the

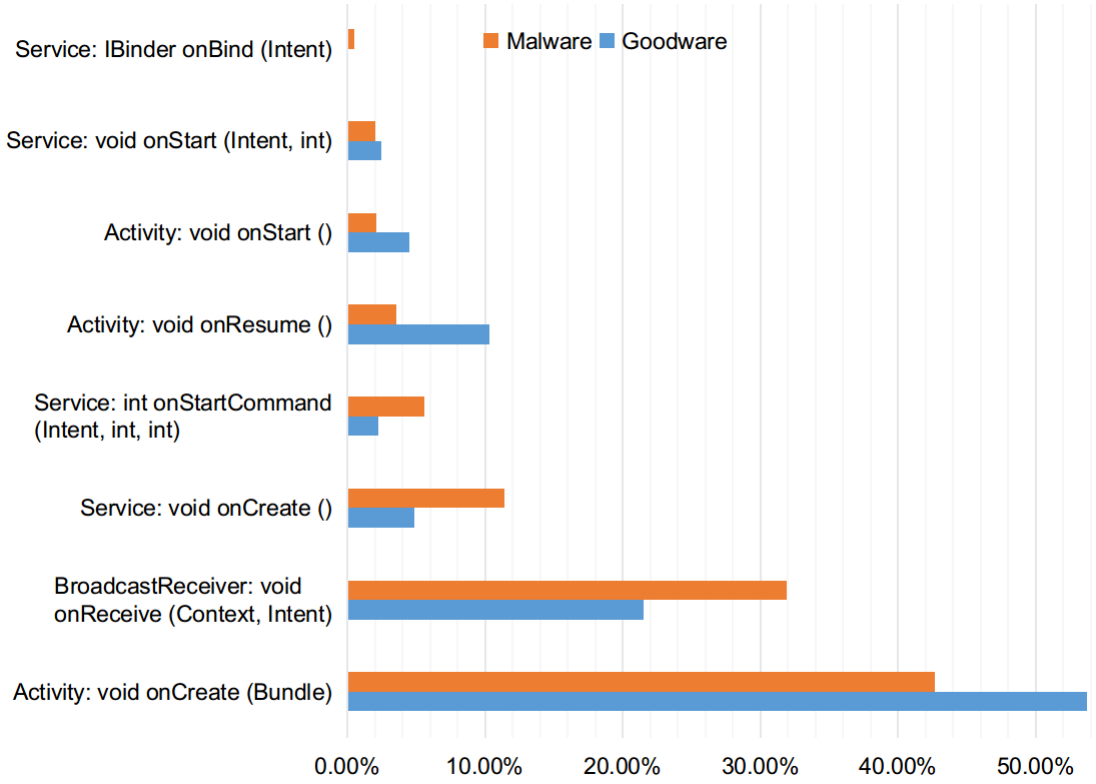

FIGURE 4.3: Use of entry points to reach suspicious methods

heavy usage of `onReceive()` allows triggering malicious actions whenever the app receives an Intent broadcast like `BOOT_COMPLETED` or `SMS_RECEIVED`. These entry points constitute an easy way to add a malicious package to a benign application without huge modifications of app because the malicious code tends to be independent of the benign one [92].

4.3.3 Implicit Edges Presence

Implicit edges play an important role in order to increase the accuracy of malware detection tools that rely on CFGs. This is confirmed by the results of our study where 47.82%

(almost the half) of the reachable suspicious methods are reachable **only** through implicit interprocedural calls. More globally, 72.69% of malware have at least one suspicious piece of code hidden behind implicit calls without any alternative execution path. These results confirm the necessity of including implicit interprocedural calls in the phase of building application CFGs, since they almost doubled the number of reached suspicious methods in our analyzed malware dataset. Obviously, an analysis tool that relies on application CFGs to reach targeted code without taking into consideration this type of calls could miss a significant part of the malicious behavior.

We have also focused on the nature of implicit calls, and we discovered that one of the most used implicit calls is due to the pair of registration-callback: `(Thread.start(), Runnable.run())`. Such callbacks are used to launch a thread from the main application. They can be used to perform heavy asynchronous tasks like downloading data from Internet or encrypting files, which may slow down activities and affect the user experience or force Android to kill the application. We find also that many callbacks from the `Handler` class are used. A `Handler` can be used to schedule messages and runnables to be executed later and to enqueue an action to be performed on a different thread. Once again, using an handler helps create an execution separated from the main thread. We also found a callback of a different nature: `(setOnClickListener(), onClick())` which is related to elements of the graphical interface suggesting that malware may be triggered by actions performed by the end user.

4.3.4 Triggering Conditions

Android malware use a clever technique to hide their harmful behavior. They use conditions along the execution path to deviate any runtime analysis tool from reaching the malicious code. For instance, we found in this experiment an average of 12.34 conditions per path leading to suspicious code location. These conditions are a mix of necessary checks for the app to work, and triggering conditions that prevent the malicious code from running except in specific situations. Most of the conditions that are in the execution paths are just ordinary, nevertheless some of them are interesting from a security point of view. Indeed, some conditions are used by malware to trigger malicious actions. Listing 4.3 shows a triggering condition example where the *IMEI* of the device is sent to a remote server if the incoming system event is due to an SMS reception. This condition is extracted from the malware sample presented in section 4.1.4.

```
public void onReceive(Context paramContext, Intent intent) {
if(!intent.getAction().equals("android.
    provider.Telephony.SMS_RECEIVED")) {
/* ... */
localArrayList.add(new BasicNameValuePair("imei", "IMEI"));
new HttpSend("http://up.#######.com/", localArrayList).execute(new
    Integer[0]);
}
/* ... */
}
```

LISTING 4.3: Triggering condition

We will return to the triggering conditions issue in Chapter 5 to show in details how malware use them to prevent its code from being detected by automatic analysis environments.

4.4 Comparing with Benign Apps

In order to compare the findings obtained by studying the malware dataset, we analyzed a collection of 2,311 goodware samples provided by AndroZoo [93]. This helps to emphasis the difference between the characteristics of malicious and benign applications. We did the same analysis on these benign applications as the one performed on the malicious ones. The analysis took an average time of 86.29 seconds per app, and the apps were of an average size of 80 Kb.

The results are depicted in blue in Figure 4.2, where it shows the usage of sensitive API calls in the analyzed benign application set. We note that malicious applications use overall more suspicious calls than the benign ones. However, the proportions are bigger for suspicious API calls such as those used for encryption. These methods are often used by malware to decrypt binary code in order to load it dynamically and to encrypt personal data before sending it to remote servers.

The use of entry points is depicted in blue in Figure 4.3. The main information that we can extract from this figure is the difference in usage of `BroadcastReceiver`:

`void onReceive (Context, Intent)` between benign and malicious apps. As mentioned before, malware rely a lot on system events to launch malicious actions unbeknownst to the user.

4.5 Discussions

The conducted experiments have shown the importance of having useful CFGs that include implicit calls for malware security tools. To connect different method CFGs, GPFinder uses API summaries generated by EdgeMiner which is built for Android version 4.2. Thus, for a better results, it should be updated. However, as we showed in Section 4.3.3, the most used implicit calls are related to multitasking and message exchange, which have not changed a lot since Android 4.2 as far as we know.

Our experiments show that we can almost double the coverage of suspicious code by including implicit calls while building global CFGs, although, there is no other accurate implicit calls tool to compare GPFinder to. Thus, we do not have statistics about the accuracy of our tool, but it depends on the used summaries, in this case the EdgeMiner's ones.

Implicit calls can easily be used by Android malware to hide their code. This is not specific to Android malware nor to Android, but it is a feature of the Java language. However, Android heavily uses event-driven callbacks, a characteristic that can be easily exploited by malware authors.

4.6 Conclusion

This chapter presented GPFinder, a practical solution to help security experts to understand and analyze Android malware. GPFinder determines the suspicious code locations in Android applications. Then, for each method in the bytecode considered as suspicious, GPFinder exhibits all execution paths that start from an entry point and lead to that method. For this purpose, GPFinder is the first approach able to take the Android framework itself into account by computing a global control flow graph with implicit edges related to the callback mechanism.

We have evaluated, on a collection of 14,224 Android malware samples, how implicit interprocedural calls are used by malware. Our experiments show that 72.69% of

malware have at least one suspicious piece of code hidden behind implicit calls without any alternative execution path. We demonstrated that we can almost double the coverage of suspicious code by including implicit calls while building global CFGs. Our experiments have shown that conditional statements are used along the executions paths. Some of them are used specifically to prevent the suspicious code from running and thus escaping detection by runtime analysis tools. We will discuss this particular issue in details in the next chapter.

Chapter 5

Triggering Suspicious Code

In the previous chapter, we tackled the question of execution paths in Android applications. We showed that control flow graphs should contain the necessary paths towards the suspicious code, and we demonstrated its importance for runtime analysis tools. We proposed to complete CFGs by adding implicit interprocedural calls. In this chapter, we explore the issue of launching the suspicious code inside Android applications automatically, and how we use our previous work on execution paths to achieve this.

Some previous works have already worked on the subject of triggering Android malware behaviors. They can be classified into two categories depending on how they handle the triggering protections. The first category of approaches solely stress the user interface or change the underlying environment by injecting framework events and/or intercepting API calls and modifying them [12], [17], [18]. This approaches fail to trigger certain complex logic bombs. For instance, when the malware uses a hash function to process an input and compares it to an internally stored value. Obviously, it becomes tedious to guess the input to provide in order to get the right hashed value and thus bypass the triggering condition. The second category of triggering approaches aim at executing some slices of code without preserving the application context [20]. Such techniques can be useful for testing purpose, but they can expose an unrealistic behavior by removing part of the code that are necessary to understand the overall malicious behavior. This partial view of the app's behavior can mislead the analysis tool and extract partial or wrong information. For example, sending data over the network is not a malicious action *per se*, but writing the localization coordinates history to a file and sending it to a remote server once the phone is online could be considered as a malicious activity. Thus, such approaches can cause a loss of the original malicious context.

We believe that a hybrid approach would obtain at least, similar triggering results

while preserving the context of the original malicious code. Events that start the execution paths are still required, but we think that the conditions that protect the malicious code can be influenced by overloading the variables that are associated to them. With such a strategy, we avoid complex computation of objects or inputs (like Android Intents). We propose in this dissertation to trigger as much as possible the suspicious code while doing our best to preserve the application context i.e. the rest of the code that encompass the suspicious code. Our approach uses a combination of static and dynamic analysis techniques. The former are used to compute the control flow paths that enable to reach the identified suspicious basic blocks. This analysis also collects the triggering conditions that will be involved at runtime on the identified control flow paths. Then, we introduce data dependency techniques for extracting constraints on the variables that are involved in the triggering conditions. Possible values of the variables are then computed with an SMT solver. We propose to slightly modify the variables involved in the triggering conditions in the malware bytecode in order to push the execution towards the suspicious code.

We implemented our approach in a tool called TriggerDroid. It works directly on the application's bytecode, without the need of the source code. Our experiments use a dataset of 135 state-of-the-art malware samples (different varieties), covering 71 different malware families [21]. Results show that our approach, while promising in suspicious code triggering, needs more refinement and adaptation to handle special cases due to the highly diverse malware dataset that we analyzed. Our evaluations quantify how much the supposed suspicious code of malware samples can be successfully executed using this approach. We also present the performance results compared with other approaches, while analyzing a highly diverse malware dataset.

This chapter is structured as follows. First, Section 5.1 introduces a motivating example that explains why executing malicious code behind triggering condition is a difficult task. Section 5.2 presents the static analysis part of our approach and Section 5.3 explains how the suspicious code is triggered. In Section 5.5, we detail our implementation, and present the results that we obtained. We discuss our approach and present our future work in Section 5.6 and, finally, conclude the chapter in Section 5.7.

5.1 Motivating Example

In this section, we present a motivating example that shows the importance of carefully considering the triggering conditions when the goal is to execute the payload of a

malware.

Listing 5.1 gives an example of malware that starts when an SMS is received. It checks the content of the message and, depending on of the result, it decides whether it sends back the location of the device or not. In this example, there exist many conditions that must be satisfied in order to execute the code that sends the location back. In order to trigger the slice of code starting at line 35, the application must receive an `SMS-RECEIVED` intent (communication message in Android terminology), which triggers the `onReceive` method beginning at line 2. First, the code checks if the smartphone is not an emulator (line 5). Then, it verifies if the received intent's action is indeed due to an SMS reception (line 9). To extract the received message and sender number, the app checks if the intent's bundle is not null at line 11. Then, the app verifies if the PDU object i.e. the content of the message is not empty at line 15. Then it verifies if the hash of the message content equals an internally stored value (line 26). Finally, at line 31, the malware waits for 1 hour before sending the location back to the sender number.

In this listing, the malicious code (lines 35 to 38) is protected by several conditions that are not straightforward to trigger. These conditions are of different natures: some check an intern value (like the hash of message content), others check the environment (like the intent's action). The malware also stay dormant for a while to escape any short analysis runtime environment.

Randomly fuzzing the application (the two parameters context and intent) would have a slight chance to trigger the malicious behavior, because the app checks the hash of a message. Relying only on external fuzzing cannot trigger the malicious code of this application because of the MD5 sum comparison. Obviously, reversing the hash function is virtually impossible. On the other hand, relying solely on application internal modification cannot execute the malicious code without heavy modifications or crashing the application. It should receive a valid SMS in order to extract the message and the sender number.

In the next sections, we present our approach to cope with this triggering problem. We aim throughout this dissertation at triggering the malicious code while preserving the most important information about the suspicious code and its context.

5.2 Prior static analysis

Our suspicious code triggering approach, implemented in a tool called TriggerDroid, follows these four steps.

```
String SMS_RECEIVED = "android.provider.Telephony.SMS_RECEIVED";
public void onReceive(Context context, Intent intent) {

   // Check if the device is an emulator
   if(isEmulator())
      return;

   // Make sure the received intent is due to an SMS reception
   if (intent.getAction().equals(SMS_RECEIVED)) {
      Bundle bundle = intent.getExtras();
      if (bundle != null) {

         // Get message and sender
         Object[] pdus = (Object[]) bundle.get("pdus");
         if (pdus.length == 0) { return; }

         SmsMessage[] messages = new SmsMessage[pdus.length];
         StringBuilder sb = new StringBuilder();
         for (int i = 0; i < pdus.length; i++) {
            messages[i] = SmsMessage.createFromPdu((byte[])
                pdus[i]);gitageBody());
         }
         String num = messages[0].getOriginatingAddress();
         String message = sb.toString();

         // The message should be "collect"
         if(md5sum(message).equals("0788a6922bd5f9f130e7ed8980193bab")){

            // Stay dormant for 1 hour
            long duration = 3600000L // 1 hour
            try{
               Thread.sleep(duration);
            } catch(InterruptedException e){return;}

            // Send the location back
            tm = getSystemService (Context.TELEPHONY_SERVICE);
            String location = getLocation();
            SmsManager sm = SmsManager.getDefault();
            sm.sendTextMessage(num, null, location, null, null);
            }
         }
      }
   }
}
```

LISTING 5.1: Motivating example for TriggerDroid

1. locate all suspicious sites s in the bytecode;
2. For all s, select an execution path p towards s that minimizes the number of triggering conditions;
3. Alter the bytecode to push the control flow along the path p;
4. Run p.

These steps will be explained in detail in the next sections.

5.2.1 Identifying the Suspicious Basic Blocks

TriggerDroid transforms the Dalvik bytecode (`.dex` files) of the processed application into Jimple [94], a stackless intermediate representation of the bytecode. The Jimple language uses a 3-address representation where most instructions use the template `x = y op z` with `op`, one of 19 available operations in Jimple. `x,y,z` are typed and explicitly declared variables, prefixed with a $ symbol. For the bytecode interpreter or the compiler, they represent stack positions. Lastly, the control structures `for` and `while` are replaced with simple `if` and `goto` statements. A running example code is given in Listing 5.2 and the corresponding Jimple code is given in Listing 5.3.

TriggerDroid identifies some basic blocks as suspicious using a scoring algorithm based on the evaluation of the API classes [79]. We have already explained this in Section 4.1.2. The score points out the basic blocks that invoke at least one method of a potentially suspect API as for instance `javax.crypto.android.telephony.TelephonyManager` or `android.net.Network`. For example, TriggerDroid identifies as suspicious the line 13 of Listing 5.2 as it relies on the use of `android.telephony.SmsManager`. In this listing as in the rest of the figures or listings, suspicious code is highlighted in red.

5.2.2 Triggering Conditions and Variables

As we saw earlier in this chapter, Android malware can implement triggering conditions in order to evade possible dynamic analysis. In our motivating example of Listing 5.1, the malware has three triggering conditions: it tests if the device is an emulator, if an SMS has been received and if the hash of its content equals a certain stored string. As TriggerDroid intends to execute the payload, it extracts the triggering conditions in the

```
1  void h() {
2      int w, x, y, z;
3      w = -f();
4      x = f();
5      if (x > 2) {
6          y = 2 * x;
7          if (y > 10) {
8              z = g();
9              if (z > 0 && z == w) {
10                 return;
11             }
12             SmsManager manager = SmsManager.getDefault();
13             manager.sendTextMessage("1234", null, String.valueOf(x),
                   null, null);

14
15         }
16     }
```

LISTING 5.2: Running example Java code

```
1  void h()
2  {
3      com.example.dexman.eg2.MainActivity $r0;
4      int $i0, $i1, $i2;
5      android.telephony.SmsManager $r1;
6      java.lang.String $r2;
7      $r0 := @this: com.example.dexman.eg2.MainActivity;
8      $i0 = virtualinvoke $r0.<com.example.dexman.eg2.MainActivity: int
           f()>();
9      $i0 = neg $i0;
10     $i1 = virtualinvoke $r0.<com.example.dexman.eg2.MainActivity: int
           f()>();
11     if $i1 <= 2 goto label2;
12     $i2 = 2 * $i1;
13     if $i2 <= 10 goto label2;
14     $i3 = virtualinvoke $r0.<com.example.dexman.eg2.MainActivity: int
           g()>();
15     if $i3 <= 0 goto label1;
16     if $i3 != $i0 goto label1;
17     return;
18   label1:
19     $r1 = staticinvoke <android.telephony.SmsManager:
           android.telephony.SmsManager getDefault()>();
20     $r2 = staticinvoke <java.lang.String: java.lang.String
           valueOf(int)>($i1);
21     virtualinvoke $r1.<android.telephony.SmsManager: void
           sendTextMessage(java.lang.String, java.lang.String,
           java.lang.String, android.app.PendingIntent,
           android.app.PendingIntent)>("1234", null, $r2, null, null);
22   label2:
23     return;
24 }
```

LISTING 5.3: Running example Jimple code

Jimple representation of the bytecode. The `if` and *switch* statements are extracted using a part of the grammar of Soot [35]:

ifStmt	$\longrightarrow$	`if` *conditionExpr* `goto` label ;
conditionExpr	$\longrightarrow$	imm_1 *condop* imm_2 ;
condop	$\longrightarrow$	$\leq \mid \geq \mid = \mid \neq$;
imm	$\longrightarrow$	variable | constant ;

We define as Triggering Condition (TC) an `IF` statement that may drive the execution away from the payload of the malware. For a given path p of the control flow graph that starts from an entry point and reaches a suspicious basic block, a triggering condition is an `IF` statement belonging to the path where either its `then` or `else` branches end outside the execution path. Formally, a node n is a triggering condition for the path p `iff` n is an `IF` statement and its post-dominator also belongs to the path p. Obviously, an `IF` statement that is not in the path cannot be a triggering condition. An `IF` statement that is in the path but whose post-dominator is also in the path is noted "Non TC", because the control flow stays in the path regardless which branch the execution takes. Finally, we call *triggering variables* the variables used by at least one triggering condition. In Listing 5.3, `$i1` and `$i2` are triggering variables.

Figure 5.1 sums up the different triggering conditions that TriggerDroid has to cope with. Our running example (function $h()$) is represented on the bottom left part of the figure. An execution path, p ,is highlighted in pink from the entry point `A.onCreate()` to the suspicious basic block (red node). On the path p, most of the nodes are triggering conditions (TC) because they can steer the flow out of the path. The two encountered nodes of the activity `A` are not triggering conditions because all their branches stay in p.

Triggering conditions play an essential role to steer the code towards the suspicious basic blocks. They can be identified once the paths that leads to the suspicious code has been extracted.

5.2.3 Path Computation

TriggerDroid computes execution paths towards the suspicious basic blocks based on the interprocedural control flow graph (CFG) computed from the Jimple representation of the bytecode as explained in Chapter 4. This CFG represents the possible execution paths of the whole application including the implicit transitions that may be due to calls that pass though the Android framework.

For a given suspicious basic block whose s is the first statement, if $\mathscr{E}$ denotes all entries points of the application, TriggerDroid returns a path p from an entry point $e \in \mathscr{E}$

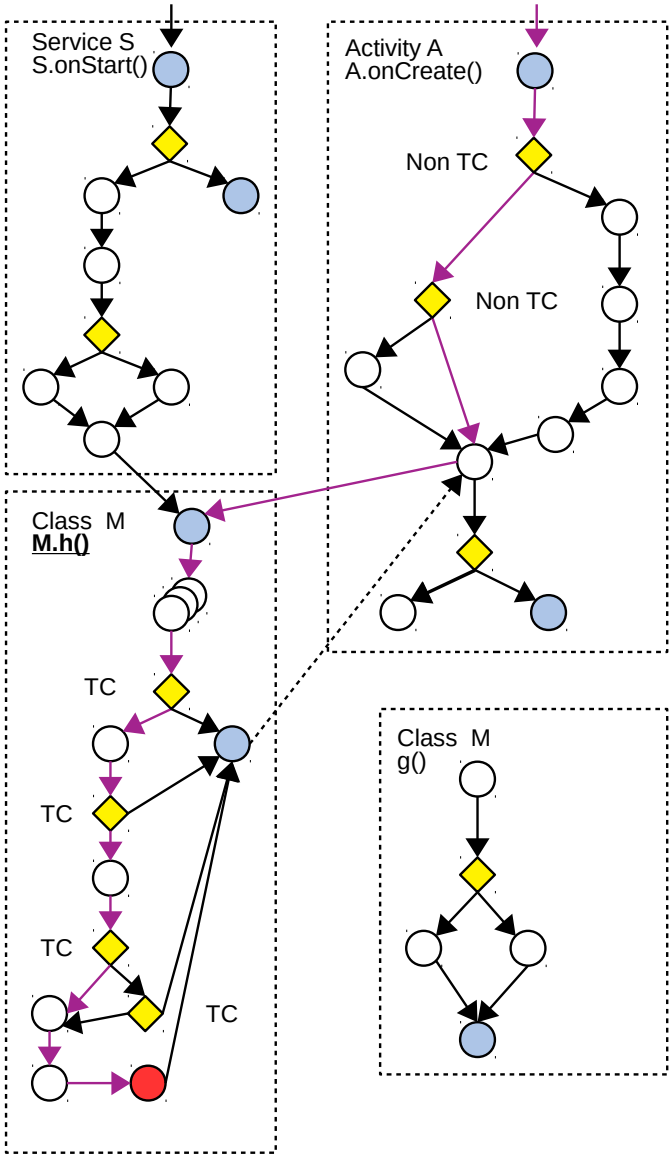

FIGURE 5.1: Inter-procedural CFG of the running example

towards s. While we base our path search on GPFinder, we do not aim at finding all possible execution paths towards a given suspicious basic block. We only search a path that minimizes the number of modifications to be done on the application code. TriggerDroid implements a variant of the classical path search algorithm $\mathscr{A}^\star$ to compute a path that appears to be the quickest one that lead to the suspicious code with a minimal cost. Intuitively, TriggerDroid searches for the path with few triggering variables. It explores a set of possible paths by backward computation and returns the one that minimizes the cost function $f(n) = g(n,p) + h(n)$ where $g(n,p)$ denotes the cost function to join the node n in the current path p being computed, and $h(n)$ denotes an estimation function of the cost to join an entry point $e \in \mathscr{E}$ from the node n. Intuitively, the cost $g(n,p) = \sum_p cost(n,p)$ increases with the difficulty to cope with the triggering conditions to reach the node n in the current path p. The scores have been chosen to prioritize paths that minimize the alterations to be done on the application.

$$
cost : \left| \begin{array}{lcll}
(n,p) & \longmapsto & 1 & \text{if } n \text{ is not a triggering condition} \\
(n,p) & \longmapsto & 30 & \text{if } n \text{ is a triggering condition} \\
 & & & \text{dealing with one variable} \\
(n,p) & \longmapsto & 50 & \text{if } n \text{ is a triggering condition} \\
 & & & \text{dealing with two variables}
\end{array} \right.
$$

The cost function $g(n,p)$ expresses that the cost of a simple instruction is negligible compared with a triggering condition. It expresses also that it is less difficult to cope with a triggering condition dealing with two variables than with two triggering conditions dealing with one variable each.

The cost estimation $h(n)$ to join an entry point $e \in \mathscr{E}$ from a node n is defined as follows. $h(n) = 0$ if n belongs to an entry point method (as `OnStart()`, `OnCreate()`, `OnReceive()`, ...). $h(n) = 5$ if n is not in an entry method but it is part of a class that contains an entry point method. If n belongs to a class without entry point, $h(n) = 10$. The main difference between our implementation and the classical implementation of $\mathscr{A}^\star$ is that the cost of a node $g(n,p)$ may vary depending on the path being computed. In addition, our algorithm searches a path towards any entry point. Thus, it may have more than one final node to chose from.

For our running example (Listing 5.2 and 5.3), TriggerDroid computes the path p represented in pink. The cost of the path p (cf. Figure 5.1) is 92 because 3 triggering conditions are involved in $h()$ and 2 non triggering conditions are involved in A. The extracted triggering conditions on the identified path p are given in Listing 5.4. These

```
Triggering conditions to reach the target:

Triggering condition 1 with one triggering variable ($i1)
if $i1 <= 2 goto staticinvoke
<android.util.Log: int i(java.lang.String,java.lang.String)>
in <com.example.dexman.eg2.MainActivity: void h()>

Triggering condition 2 with one triggering variable ($i2)
if $i2 <= 10 goto staticinvoke
<android.util.Log: int i(java.lang.String,java.lang.String)>
in <com.example.dexman.eg2.MainActivity: void h()>

Triggering condition 3 with one triggering variable ($i3)
if $i3 <= 0 goto staticinvoke
<android.util.Log: int i(java.lang.String,java.lang.String)>
in <com.example.dexman.eg2.MainActivity: void h()>
```

LISTING 5.4: Triggering conditions for the running example

conditions check the value of one variable each ($i1, $i2 or $i3).

5.3 Automatic Triggering

Once TriggerDroid has statically computed an execution path p that joins a suspicious code location, it slightly alters the bytecode of the application in order to force the execution to stay in p.

5.3.1 Triggering Strategies

TriggerDroid injects in the code new statements that update the values of the triggering variables; i.e those that appear in the triggering conditions. These new statements drive the execution when a triggering condition is encountered and keep the execution on the path. The new values are assigned to variables to satisfy the triggering conditions. We point out three triggering strategies:

Strategy 1: All triggering conditions are replaced by `GOTO` statements. This was already implemented in GroddDroid [19]. It is a simple strategy that tries to force the execution of the desired conditional branches independently of the execution context. This may lead to situations where the variables' values become inconsistent for the operations to be executed and it makes the application crash.

Strategy 2: Assignment statements `v = rvalue` are inserted *right before* the corresponding triggering condition. This helps to keep the execution environment consistent with the execution. It adds new assignment statements in the bytecode that override the previous values of variables occurring in the triggering conditions. Values are chosen to keep the execution along the desired path and the new assignments are added just before the concerned triggering condition. For example on Figure 5.3, if we insert the assignments $\$i1 = 3$, $\$i2 = 11$ and $\$i3 = 0$ one by one before each triggering conditions, the execution will take and stay on the path with the pink color.

Strategy 3: Assignment statements `v = rvalue` are inserted *after* the reaching definition of the corresponding triggering variables. This helps to limit the modifications done on the malware to what is strictly necessary and increase the consistency. Strategy 3 also adds new assignment statements in the bytecode. Only the location of these new statements differs from Strategy 5.3.1. The second strategy does not care of data dependencies, it modifies variables values appearing in the triggering conditions whereas these variables may depend on each others which may leads to inconsistencies in the execution context. On the contrary, Strategy 5.3.1 takes into consideration the triggering conditions that are influenced by common variables involved in a chain of computation. In this case, Strategy 3 adds the required new assignments just after the definition of the variables. In the bytecode, a variable can be used much later after it has been defined. An instruction defines a variable, when it declares, initializes or changes the value of this variable. An instruction uses a variable when it employs this variable in other manner. If a variable v is defined at point T belonging to an execution path that later reaches a node U where v is used without being overwritten by another value in the path, then (T, U) forms a *definition-use* pair and v is a reaching definition at U. Definition-use pairs record direct data dependencies and form a data dependency graph. TriggerDroid relies on a subgraph of the data dependency graph where nodes defining variables by method invocation or any expressions using more than one variable are not linked to the others.

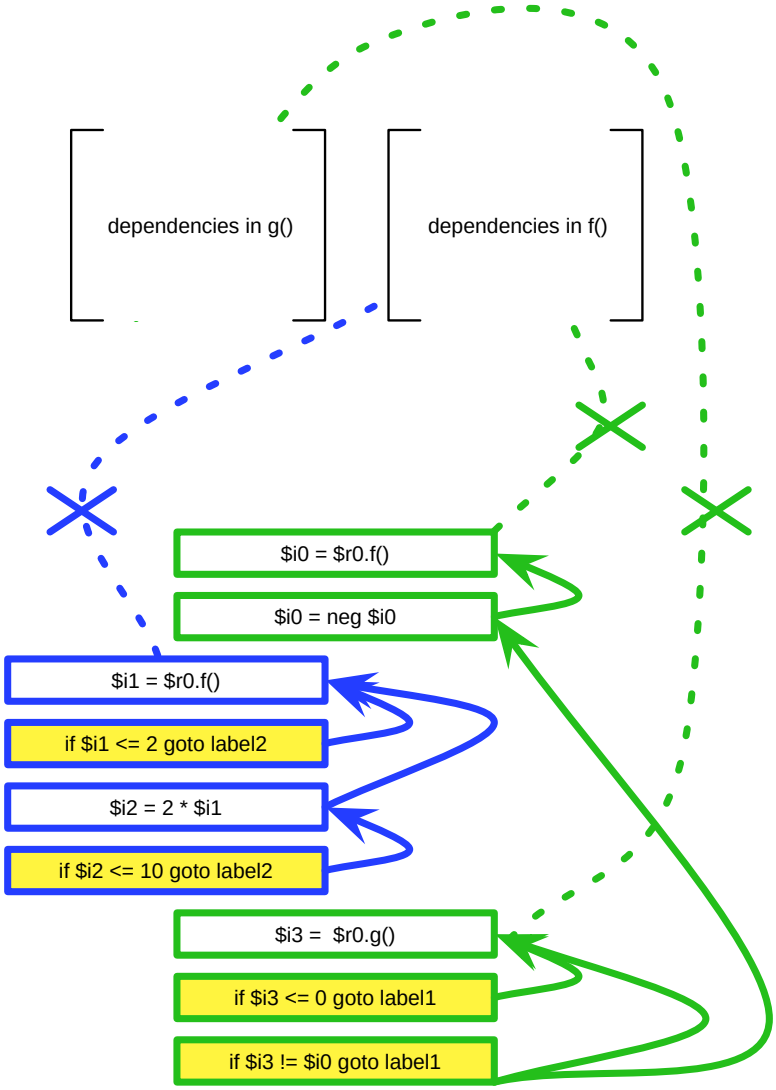

FIGURE 5.2: Data dependency subgraph used by TriggerDroid

Figure 5.2 depicts the data dependency subgraph used by GroiddDroid on our running example. For the statement `if $i2 <=10 goto label2`, TriggerDroid finds the following data flow: `if $i2<=10 goto label5` → `$i2 = 2* $i1` → `$i1 = $r0.f()`. For example, the definition in statement `$i1 = $r0.f()` is a reaching definition of `$i1` at statements `if $i1 <= 2 goto label2` and `$i2 = 2 * $i1`.

TriggerDroid uses connected component of this data dependency subgraph depicted in green and blue in Figure 5.2. Each connected component of the data flow is a group of triggering condition where a variable may influence the triggering conditions. For example, conditions `if $i1 <= 2 goto label2` and `if $i2 <= 10 goto label2` are in the same connected component since they have a common data dependency (reaching definition in `$i1 = virtualinvoke $r0.f()`).

5.3.2 Satisfying Triggering Conditions

Triggering conditions should be satisfied by modifying the values of their variables according to the chosen strategy. A triggering variable occurs in an `if` statement on the form `x op y` where `x`, `y` are variables or constants and `op` belongs to $\{\leq;\geq;=;\neq\}$. Consequently, the constraints can be represented as combinations of linear equation on variable representing primitive types (integers, floats, booleans and strings). The same remark applies for the `switch` statements. Thus, TriggerDroid can pass the constraints to an external SMT solver to find the variables' values that satisfy them.

Strategy 2 takes each TC one by one, and solves them independently of the others. The output is a set of variable values for each triggering condition. For instance, the solver returns $\$i1 = 3$, $\$i2 = 11$ and $\$i3 = 0$ for the running example. These values are inserted just before the triggering conditions, as represented in Figure 5.5.

For strategy 3, each statement group of a connected data flow component generates a constraint group, and thus an independent SMT program. Each node of the connected components generate a constraint, except function calls. For example, the SMT program #1 (resp. #2) of Listing 5.5, corresponds to the upper (resp. lower) connected component of Figure 5.2. In the first SMT program, only one triggering condition is crossed by the pink path of Figure 5.3. Thus, `$i3 != $i0` and `$i0 = neg $i0` are dropped[1]. In the second SMT program, triggering conditions `if $i1 <= 2 goto label2` and `if`

[1]Note that variable names have subindexes in the SMT program when a variable is killed and redefined: the constraint would be $\$i0_1 = \$i0_0$.

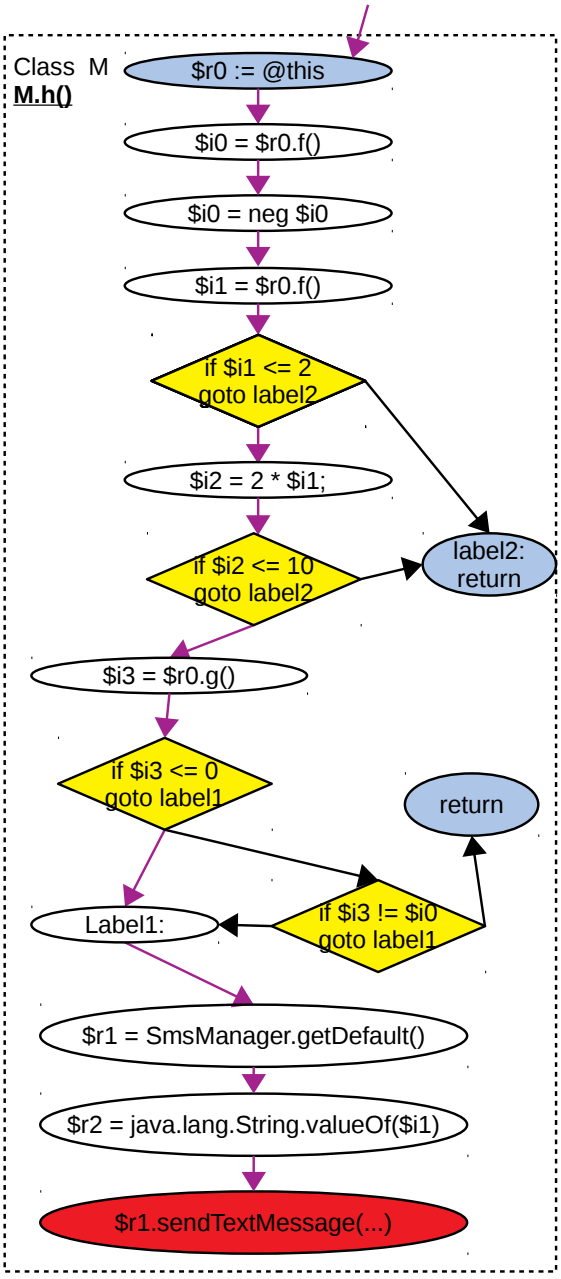

FIGURE 5.3: Running example's CFG

`$i2 <= 10 goto label2` are solved together with `$i2 = 2 * $i1`. Consequently, the SMT solver returns one variable assignment that satisfies both of them, `$i1 = 6`.

5.3.3 Delicate statements

During the static analysis, a special procedure collects "delicate statements". These statements may crash the application if a modification of their variables' values generates an exception. A simple example of such statement is $\$i2 = 2/\$i1$. In this case, $\$i1$ should be different from 0. Delicate statements are collected in order to add constraints to pass to the SMT solver when computing new values for the variables. To this stage, we have implemented only delicate statements that are related to division by zero. Other types could be null pointer references and string manipulation methods like `indexOf` and `substring`.

5.3.4 Malware Alteration and Execution

The Insertion of the new statements concludes TriggerDroid's static part. These statements are on the form `x = v` where `x` is a triggering variable and `v` is a suitable value for `x` computed by the SMT solver. In strategy 2, these new statements are put just before each triggering conditions. Comparing to strategy 2, strategy 3 modifies the variables according to the chosen place in the data flow graph. Listing 5.6 presents the bytecode of our running example when TriggerDroid applies strategy 2 and 3.

At this stage, the malware is ready to be executed by TriggerDroid for the chosen suspicious basic block. TriggerDroid runs the malware, combined with additional techniques presented in the next section that helps to simulate AOSP events and the graphical interface manipulation [19]. Then, TriggerDroid monitors the execution in order to check if the suspicious code is executed or not.

5.4 Implementation

TriggerDroid extends Soot [35] to perform the static analysis phase. It operates directly on the application bytecode represented using the Jimple language. For solving the SMT programs, TriggerDroid relies on the SMT solver Z3 [95].

In order to know if the targeted code has been triggered or not, TriggerDroid instruments the bytecode of the application in order to output the information in the logging

```
// SMT Program #1:
(declare-const $i3_0 Int)
(assert (<= $i3_0 0) )
(check-sat)
(get-model)

// Solution of program #1:
sat
(model
(define-fun $i3_0 () Int
0)
)

// SMT Program #2:
(declare-const $i1_0 Int)
(declare-const $i2_0 Int)
(assert (not (<= $i1_0 2 ) ) )
(assert (= $i2_0 (* 2 $i1_0 ) ) )
(assert (not (<= $i2_0 10 ) ) )
(check-sat)
(get-model)

// Solution of program #2:
sat
(model
(define-fun $i1_0 () Int
6)
(define-fun $i2_0() Int
12)
)
```

LISTING 5.5: Constraints on triggering variables sent to the SMT solver

```
1  void h()
2  {
3  com.example.dexman.eg2.MainActivity $r0;
4  int $i0, $i1, $i2;
5  android.telephony.SmsManager $r1;
6  java.lang.String $r2;
7  $r0 := @this: com.example.dexman.eg2.MainActivity;
8  $i0 = virtualinvoke $r0.<com.example.dexman.eg2.MainActivity: int
       f()>();
9  $i0 = neg $i0;
10 $i1 = virtualinvoke $r0.<com.example.dexman.eg2.MainActivity: int
       f()>();
11 $i1 = 3; // Strategy 2
12 $i1 = 6; // Strategy 3
13 if $i1 <= 2 goto label2;
14 $i2 = 2 * $i1;
15 $i2 = 11; // Strategy 2
16 if $i2 <= 10 goto label2;
17 $i3 = virtualinvoke $r0.<com.example.dexman.eg2.MainActivity: int
       g()>();
18 $i3 = 0; // Strategy 2 and 3
19 if $i3 <= 0 goto label1;
20 if $i3 != $i0 goto label1;
21 return;
22 label1:
23 $r1 = staticinvoke <android.telephony.SmsManager:
       android.telephony.SmsManager getDefault()>();
24 $r2 = staticinvoke <java.lang.String: java.lang.String
       valueOf(int)>($i1);
25 virtualinvoke $r1.<android.telephony.SmsManager: void
       sendTextMessage(java.lang.String, java.lang.String,java.lang.String,
       android.app.PendingIntent, android.app.PendingIntent)>("1234", null,
       $r2, null, null);
26 label2:
27 return;
28 }
```

LISTING 5.6: Running example with strategies 2 and 3

system (Logcat). TriggerDroid also monitor the app crashes by filtering the logs generated by Dalvik VM crashes.

Data Dependency. TriggerDroid computes the data dependency of each method in application. Then, it computes interprocedural dependency. There are two types of data dependency between methods. The first one is due to arguments passed to the method at the call site. All statements that depend on an argument in the called method depend also on the corresponding argument at the call site. The second type of interprocedural data dependency is due to the method return. Any statement that depends on a return of called method at a call site depend also on the return value in the body of the called method. This way, we make sure that the data inside the application is interdependent even between different methods.

Upfront Triggers. In some cases, triggering a basic block in Android applications depends on the triggering of another code somewhere else in the application. For instance, broadcast receivers can be declared statically in the application's manifest to receive events and launch their `onReceive` methods. Nevertheless, they can also be registered dynamically in the code of the application. We cannot trigger a non registered broadcast receiver. For this matter, TriggerDroid detects the dynamically registered broadcast receivers and launches first the code that registers them. Then, it sends the event that triggers the concerned broadcast receiver. We call this TriggerDroid's mechanism " upfront triggers".

Sleep Time Reduction. As we saw in Listing 5.1, malware can wait for a while before launching their payload in order to escape runtime analysis tools. TriggerDroid detect this evasion technique and reduce the waiting time to 50 ms if it is greater than that. It inserts a new assignment for the variable used in the `sleep` method.

Framework event injection. TriggerDroid uses a mix of methods to inject Android framework events in order to trigger the desired app component. It can inject events like incoming and outgoing calls and SMS, install and uninstall application, etc. TiggerDroid uses a vanilla Android build on both physical devices and emulators. In the case of physical devices, TriggerDroid install an helper application to stimulates framework events like making a call, or broadcasting an `SMS_RECEIVED` intent. On emulators, TriggerDroid uses the preexisting telnet method to inject these events.

5.5 Evaluation

To evaluate our malware triggering approach, we led an experiment on a state-of-the-art Android malware dataset and compared the results with some existing approaches.

5.5.1 Experimental Setup

We selected a Android malware dataset that contains 135 varieties that belong to 71 malware families, as explained in [21]. This dataset is diverse enough to represent the most known Android malware families existing at the time when we launched our experiment. We took one malware sample from each variety and performed the static analysis part of the methodology and then launched our analysis using the early mentioned three triggering strategies.

A malware sample can contain several suspicious basic blocks. Each time we find a basic block recognized as suspicious, we assign to this basic block one or more categories among: binary execution, cryptography, SMS, telephony, and networking.

5.5.2 Effectiveness in Reaching a Suspicious Code

The triggering performance of our approach is depicted in Table 5.1. The first column contains the suspicious code categories. The second column represents a launch of the application without any modification done on its code. The other columns represent a triggering strategy each. The results show that some suspicious code categories have better triggering rate than others. Strategy 3, which is our main one, has a better triggering rate than the other strategies except Strategy 1, which merely replaces `if` and `switch` statements with simple `goto`s. Nevertheless, Strategy 3 results in fewer crashes in the application after the code modification than Strategy 1. Clearly, launching the application without any code modification has the least crash rate. Strategy 3 stands between the two approaches, launching the app without any code modification and get few crashes and a low triggering rate, or highly modifying the app code and get a lot of crashes and a high triggering rate.

5.5.3 Importance of Execution Paths

We integrated our previous work, GPFinder, to build control flow graphs that include explicit and implicit interprocedural calls. This helped to obtain paths for suspicious

	Original		Strategy 1		Strategy 2		Strategy 3	
	Trigg	Crash	Trigg	Crash	Trigg	Crash	Trigg	Crash
SMS	21.1	5.3	21.1	31.6	21.1	10.5	21.1	15.8
Binary	33.3	11.1	50.0	0.0	11.1	11.1	37.5	0.0
Crypto	25.0	25.0	33.3	33.3	0.0	25.0	25.0	25.0
Telephony	43.5	2.2	52.5	5.0	38.6	6.8	41.9	7.0
Network	16.2	8.1	32.0	16.0	22.9	5.7	25.0	9.4
Average	27.82	10.34	37.78	17.18	18.74	11.82	30.1	11.44

TABLE 5.1: TriggerDroid's performance expressed in %

basic blocks, but also for their upfront triggers if they exist. Android application often use off-the-shelf libraries to use just few packages from them. This makes a big part of the library code unreachable from the application's entry points. Even with this fact, we succeeded to find execution paths more than 60% of the existing suspicious basic blocks.

5.5.4 Efficiency

This experiment was launched on a Intel® i7 laptop with 4 cores and 16 Go of RAM. The analysis of each application is timed out after 20 minutes. This includes the static analysis, a launch without any modification on the application and the 3 triggering strategies mentioned in Section 5.3.1. For each triggering strategy and suspicious basic block, the application is launched and keeps running for 20 seconds.

5.5.5 Comparison with Other Approaches

While our triggering approach is promising, it perform at the moment of writing this dissertation less than some existing approaches like FuzzDroid, which has a triggering rate of 62.34%. This results may be explained by the high diversity of our dataset, that represent most of the existing Android malware families and their varieties. We also discovered several bugs in our implementation, and we are working to fix them. In addition, there are numerous special cases where the application crashes at the start even without any code modification, because of an incompatible API version or because of the newly added Android runtime permissions. This mechanism requires manual activation of some permissions even when they are declared in the app manifest.

5.6 Discussions and Perspectives

In this section we will talk about the limits and challenges that our approach faces. We talk also about the future work that we will conduct to improve our approach.

First, we want to emphasize that this implementation and the proposed approach do not intended to tell whether the executed code is malicious or not. It merely aims at launching the suspicious code, and it is up to the runtime monitoring tool or the analyst to decide if the sample is malicious.

By proposing TriggerDroid, we aimed at maximizing the triggering rate of suspicious code without sacrificing the original context of the malware sample. Obviously, our triggering approach has a lower triggering rate than some existing state-of-the-art approaches. TriggerDroid faces at the moment of writing this dissertation some bugs in its code. Some are due to the unstable underlying Soot static analysis framework. Another issue that faces TriggerDroid is the multitude of special cases that need careful inspection and code adaptation.

The overall code that analyzes, instruments and launched applications is working fine. Meanwhile, we are working to fix TriggerDroid's bugs and handle special cases in order to improve the triggering rate. We believe that this approach is promising and it is worth further improvement.

5.7 Conclusion

In this chapter, we proposed a novel approach to trigger suspicious code in Android applications in order to monitor their runtime behavior. In contrast to other existing code triggering approaches, ours aims at reaching high triggering rate of suspicious code while preserving the original context of the application. We implemented our approach in a tool called TriggerDroid that operates directly on the application's bytecode and does not need its source code. TriggerDroid uses a hybrid code triggering method. It crafts the necessary framework events to trigger entry points, and minimizes the modifications done on the application code to steer the execution towards the targeted code. We conducted an experiment on a set of malware from 71 different malware families and 135 varieties. Results show that our approach, while promising, needs more bug fixes and special case handling. Our future work is to improve the code quality of our implementation to cope with the highly diverse dataset that we have chosen in order

for our approach to be able to deal with the huge number of malware samples that are uploaded to Internet every day.

Chapter 6

Dilemma of Malware Datasets

During my thesis, I had to collect a lot of Android malware samples to conduct representative experiments. Indeed, up-to-date and well documented malware datasets are crucial to design better security approaches and validate new tools. Many existing works simply take random malware samples, analyze them and give a success rate. However, the experiment would not be accurate if it is performed on a few randomly selected samples. There exist a handful Android malware datasets that are dedicated for the research community. The goal of this chapter is to give a feedback on the experiments we led on malware datasets and to explain our experimental process.

This chapter is structured as follows. First, we describe our work on a set of malware that has been reverse engineered for gaining technical understanding on attack vectors. Then, we describe the most important existing datasets. Next, we present Kharon dataset, a collection of well documented Android malware. Finally, we explain our dataset choices to test our own security tools.

6.1 Understanding Malware

Understanding the nature of malware is crucial to build better security tools and test them. In [92], authors aimed at understanding techniques used by malware authors to repackage the malicious code inside benign apps. They list several useful properties that characterize samples using repackaging and help to detect them. In addition, there exist hundreds of articles online explaining new discovered malware. However, they are mostly presented for nontechnical people. They are rarely detailed enough so the researcher can get a clear idea on how the malicious action is exactly launched and how it operates. We believe that a good malware explanation should include at least these points:

- What are the malicious actions and how are they executed? With code snippets if necessary (classes, methods).
- How are they triggered? With enough details about the triggering conditions in order to reproduce them experimentally.
- How is the malicious code injected in piggybacked apps?
- Is the code obfuscated and how, if any?
- Does the malware communicate with a server, and how, if any?

6.2 Labeling Malware Families

New malware samples appear every day. With this continual evolution in the malware landscape, antivirus tools give different names or labels to the same malware on a family association. It is a challenging task to make *all* vendors agree on a single naming convention for each new malware. For instance, VirusTotal[1] uses about 58 antivirus solutions to scan malware. They give different naming for malware and some times they mislabel it. This makes automatic malware gathering for research purposes a non-trivial task. In [96] and [97], researcher have already pointed out the inconsistencies in malware labeling between different vendors. Similarly, M Hurier et al. has pointed the inconsistencies in Android malware labeling and proposed a tool called Euphony to homogenize labels coming from different antivirus tools [98]. These label inconsistencies should be kept in mind when conducting an experiment on Android malware.

6.3 Existing Datasets

Several Android malware datasets were proposed in the last years to fill the need for a ground truth where research tools and approaches can be tested and reproduced. However, they were built for different purposes and in different time spans. The following are the main ones.

[1] `https://www.virustotal.com`

Genome. This is an old Android malware dataset proposed by X. Jiang et al [99]. Its malware samples were discovered in 2010 and 2011. It contains 1,260 samples grouped in 49 families. Samples were collected from different sources such as VirusShare, Google Play and other third party security companies. Authors focus on the installation, activation, actions and permissions use of the malware. However, the exact necessary conditions to trigger the malware are not given in the paper. This dataset was public and accessible to researchers worldwide, but it is not online any more due to resource limitations. In addition, it does not represent today's Android malware landscape.

AMD. Android Malware Dataset (AMD) is a large collection of malware discovered between 2010 and 2016 [21]. It came as a solution for the obsolete Genome dataset. It contains 24,650 malware samples from 135 varieties which belong to 71 families. Samples were categorized automatically, then a few samples from each variety were manually inspected. Next, short reports that describe each malware variety were generated. However, the descriptions of malware families are very brief. They do not detail how they operate, protect their malicious code and trigger it.

Koodous. Koudous[2] is a collaborative platform dedicated to Android malware research where any registered user can upload, download, rate or add a description to an app. It is based on the social interactions between analysts over the dataset. All the apps on the platform are not necessarily malicious. They come with a report given by Androguard[3], DroidBox[4] and other scanning tools. The analysis reports are very basic and do not explain the samples nor say from which family they are, which makes it difficult to choose samples to conduct an experiment on them.

Contagio. It is a public dataset created in 2011 and updated from time to time. Malware generally is supported by a blog post and a link to downloaded it along site with a link to an article that talks about it. Contagio gives little information details about the malware and do not update its database frequently.

AndroZoo. Allix et al proposed a collection of millions of Android applications to the research community in order to engage in reproducible experiments [93]. Samples

[2] `https://koodous.com/`

[3] `https://github.com/androguard/androguard`

[4] `https://github.com/pjlantz/droidbox`

are analyzed by different antivirus tools to detect malicious ones. Authors collect apps regularly from different sources such as Google Play, Anzhi[5], AppChina[6], Genome and others. They provide an API to download apps. However, apps which are detected as malicious do not have detailed description of their actions nor how they are triggered. Later, Li et al. added a collection of pairs of repackaged and original apps to the dataset [5]. From our experience, many samples from this repackaged collection are merely adware, which is ambiguous to detect because they only add external libraries for ads that can be found on benign apps. This was confirmed by Salem et el. in [100].

DroidBench. Arzt et al. proposed a dataset to evaluate the effectiveness of taint-analysis tools on Android [6]. They implement different types of data leaks using implicit information flows, callbacks and reflection. The dataset comes with the source code of each sample. Samples are categorized depending on the data leak mechanism they use. However, samples are written by the authors and do not necessarily represent real-word malware cases. In addition, they focus on information leaks and do not address code protections nor triggering techniques.

Despite all these datasets, Android malware goes rapidly out of date due to the continually evolving attack techniques. In addition, The aforementioned datasets do not detail how exactly malware works and how it hides its code. They give little help to understand the malware's internals.

6.4 Kharon, a Well Documented Dataset

Driven by the need for a well documented Android malware collection, we proposed the Kharon dataset [23]. This dataset has been manually dissected and documented. We were able to detail the malware behavior, the used triggering techniques and the location of the malicious code. Every sample was reversed and executed on a real device, and a graph is generated that represents the interactions between objects at the OS level, like the created files, processes and network sockets. The collection contains at the moment about 19 malware families. Its construction has required a huge amount of manual work to reverse the code and analyze samples statically and dynamically. This way, the

[5] `http://www.anzhi.com`
[6] `http://www.appchina.com`

analyst can get a profound understanding of the malware samples. These are the main malware samples presented by Kharon:

BadNews. It is a remote administration tool discovered in 2013[7]. The executed malicious action depends on the commands received from a distant server, which can be downloading an APK, opening a URL, etc. It starts when a `BOOT_COMPLETED` or `PHONE_STATE` intent is received. It does not execute any malicious action before waiting for 4 hours and receiving a command from the remote server.

SimpleLocker. It is a ransomware discovered in 2014 that encrypts user files present in the SD card and asks for a ransom to decrypt them[8]. It encrypts files with AES in CBC mode with PKCS#7 padding. SimpleLocker communicates with its server through the Tor network to receive orders. The malware sends sensitive information about the user's device to its server to check later if the user has paid the ransom. It starts after rebooting the phone, then it launches tasks that run continually in background.

DroidKungFu1. It is a remote admin tool discovered in 2011 that is able to install other apps without user consent[9]. It uses different exploits to break Android security measures and escalate privileges. It starts after reception of a `BOOT_COMPLETED` intent and waits for few hours. Then, it launches several temptations to break the security system and becomes root. Next it installs a fake Google search app as a system app, which can receive remote server's commands to install, remove or start applications, or even open a web page.

MobiDash. It is an adware discovered in 2016 disguised as a card game[10]. It displays unwanted adds each time the user turns on the screen. It waits up to 24 hours after the phone restart before launching the malicious actions.

[7] `https://blog.lookout.com/the-bearer-of-badnews`

[8] `https://nakedsecurity.sophos.com/2014/06/06/cryptolocker-wannabe-simplelocker-android/`

[9] `https://www.csc2.ncsu.edu/faculty/xjiang4/DroidKungFu.html`

[10] `https://blog.avast.com/2015/02/03/apps-on-google-play-pose-as-games-and-infect-millions-of-users-with-adware/`

SaveMe. It is a spyware discovered in 2015 that presents itself as an SMS and contact list backup application[11]. It asks the user his name and phone number, and sends them to a remote server alongside with other device's sensitive information and the contact list. In addition, it sends SMS messages and makes calls to a numbers provided by the server. To spread the malware, SMS messages are sent to all the user's contact with a link to download it. Finally, to avoid its removal, the malware simply deletes its icon from the launcher so make user forget about its presence.

WipeLocker. It is a malware discovered in 2014 that wipes off the SD card[12]. It blocks some social apps with a fullscreen message stating "*Obey or be hacked*". It also sends this message to all contacts, and intercepts incoming messages and respond to them automatically with the same message. WipeLocker starts after rebooting the phone and asks for admin rights.

Cajino. It is a spyware discovered in 2015 that receives commands through the Baidu Cloud Push service. I was downloaded from Google Play more than 50,000 times. It leeks SMS messages, contacts, device information, phone number and list of all files. In addition, the malware can remove files, record sound and send SMS messages depending on the received commands.

Kharon dataset has helped us to think about advanced protection mechanisms that malware uses to hide its malicious intents. For instance, we noticed that implicit calls has a negative effect on dynamic execution because they do not appear on CFGs. Kharon is not meant to conduct large scale security tools testing. It is rather better at understanding malware details so that one can build a good detection approach. Indeed, this collection of malware is very detailed and diverse. However, malware samples goes old and new ones appear on the landscape on a regular basis. Thus, a need for a continually updated and well documented dataset persists.

6.5 Choices Made in This Dissertation

We use two datasets that we build to conduct experiments to evaluate our newly designed tools.

[11] `https://blog.lookout.com/socialpath`

[12] `http://www.virqdroid.com/2014/09/android-wipelocker-obey-or-be-hacked.html`

GPFinder. It was designed to build CFGs that include explicit and implicit calls, and provide execution paths towards targeted code. This pushed us to look for a big and diverse malware dataset to test our tool. Indeed, Kharon was a well documented dataset. However, its small size discouraged us from using it to evaluate GPFinder. We opted for using a big malware collection from Koudoos that contained more than 14k randomly selected samples because we didn't have an idea about the their families. This way, we were almost certain that the dataset is diverse enough to represent many malware families. In addition, Koodous was updating his collection regularly, which was important for us since new malware was released every day. We have published online the list of samples we used for this experiment alongside with the detailed results[13] [22].

TriggerDroid. The goal of TriggerDroid is to trigger suspicious code that may be present in Android applications. Consequently, we needed a highly diverse dataset to test if it works in most of the cases. By the time we started developing TriggerDroid, AMD was released, so we decided to use it in our experiment. To cover all possible cases in this dataset, we took a sample from each variety of in each family of malware, for a total of 135 varieties covering 71 families. This high diversity caused a lot of special cases which needed careful inspection. That kept the triggering rate of TriggerDroid relatively low, which we explained in the previous chapter.

6.6 Conclusion

In this chapter we explained the dilemma of Android malware datasets between large scale collections with few details about the samples, and small collections with highly documented reports. We pointed out the labeling inconsistency issue and discovered the most important existing datasets. Then, we presented Kharon, our well documented collection of malware. Finally, we explained why we chose certain datasets to evaluate our contributions.

[13] `http://kharon.gforge.inria.fr/gpfinder.html`

Chapter 7

Conclusion and Perspectives

7.1 Conclusion

This dissertation is focused on exposing real intents of Android application by triggering any suspicious code. First, we presented the important background information about the Android ecosystem that is necessary to understand our contributions in the field of Android malware security. We explained the Android platform, how application are put together, how they communicate with the Android operating system, and how the execution flow passes between their components.

Then, we presented the most relevant categories of malware that targets this platform. We showed the techniques that were proposed in the last years to fight against the spread of malware by understanding how it works, detecting its presence, and protecting the platform and the user from its harmfulness. In addition, we demonstrated the importance of control flow graphs to many Android malware analysis tools. We showed that the quality of these CFGs is crucial for the security tools that are based on them. Furthermore, we tackled the question of runtime analysis of Android malware. More precisely, why triggering the suspicious code can be crucial to detect any harmful intention that malware authors may hide in their applications?

Next, we highlighted the importance of Control Flow Graphs in order to analyze Android application. We showed that CFGs built for Android applications may be incomplete because of the heavy use of Android callbacks. We also presented our tool, GPFinder, that is able to build CFGs for Android applications taking into account Android callbacks and implicit interprocedural calls. It also finds execution paths towards suspicious code locations, which allows launching them if needed.

After, we investigated the techniques of preventing the malicious code from executing in runtime analysis environments, which are used by Android malware to escape

detection by automatic vetting tools. We showed the importance of exposing the real intent of Android applications in order to detect malicious behaviors and protect the end user from harmful programs. We presented our tool, TriggerDroid, that is able to take an Android application, trigger the suspicious code locations and expose its real behavior. Unlike other existing approaches, TriggerDroid aims at exposing the application malicious behavior and preserving the app's original context at the same time.

Finally, we discussed the different datasets that were used in our experiments, and we presented the Kharon dataset, a detailed manually reversed dataset.

7.2 Perspectives

7.2.1 Short Term Perspectives

The triggering rate of TriggerDroid is low compared with some other existing approaches. We have several possible tracks to patch it and improve the triggering of suspicious code. This includes:

Framework events We rely on framework events to launch entry point methods. This includes broadcasting intents like `SMS_RECEIVED` and `BOOT_COMPLETED`, and directly starting some components, which is the case for activities and services in particular. If for some reason the targeted component does not start, when the intent broadcasting requires an admin privilege for instance, we will have a small chance to trigger the suspicious code.

Condition altering We modify conditions by inserting new variable assignments before them depending on the selected strategy, as explained in Section 5.3.1. We do take delicate statements that may crash the app into consideration. However, we check only statements that perform division to avoid division by zero. In addition, we do not check the possible negative effects of our alteration in other methods. This could also be a possible cause of the low trigger rate.

CFG completeness Our CFGs are generated only from the bytecode present in the `.dex` file. We do not analyze any dynamically loaded bytecode nor native code. This could have a negative impact on the completeness of the CFG. Consequently, some suspicious code could be unreachable. For example, when attackers use automatic packers that hide the payload by packing it and then unpacking it at runtime.

Over approximated paths While studying information flows in Linux systems, L Georget et al. pointed out that some control flow paths are impossible to take, because of the contradictory conditions they have [101]. For instance, if we check `if(b)` to do some task, and later we check again if(b), it would be impossible to take the path `if(b) then` → `if(b) else` without modifying the value of `b` in between. We could have similar situations in Android programs. Thus, this could be a possible factor of keeping our triggering rate low.

Bugs in TriggerDroid We are already aware of the presence of some bugs in the implementation of TriggerDroid. However, some bugs are tedious to track because they do not crash the analysis program. In addition, these only happen on few samples. Thus, they are difficult to reproduce on small code and require to investigate the full code of the faulty sample. We are actively working on this track.

Buggy malware Because of easiness of developing and publishing Android malware nowadays, we found some samples that are very badly developed. Sometimes they crash at the start. With many samples, it could be nontrivial to know if the triggering method does not work or it is because of the sample.

Blocking code Some times, a method that takes a while is called in the middle of the selected execution path. This can prevent the targeted code from launching in a short analysis time. TriggerDroid does not handle this case which could participate in preventing suspicious code triggering.

Complex objects Until now, we handle only simple variable types, such as integers, boolean, strings .etc. However, some conditions depend on complex objects. Take the example of if(A.equals(B)) with `A` and `B` two complex objects. This is translated into Jimple approximately like this: `$b1=A.equals(B); if ($b1) then goto labelX`. To take the *"then"* branch; TriggerDroid forces the condition by inserting right before it this assignment `$b1=true;`. This indeed bypasses the condition, but it may cause negative effects later in the execution path since `A` and `B` are not *equal*.

We are currently investigating all the aforementioned tracks to improve the triggering rate of our approach.

Sometimes when we do not succeed to trigger malicious code in a large scale dataset, we find it difficult to know if the sample is not functioning at all because of a bad

development or an incompatibility, or it is because our method fails to trigger it. This may seems trivial for few samples where we can manually check them. However, when the dataset gets bigger, an automatic approach is needed to cope with it. A possible track could be to find an optimal environment where the malware runs without any crash, then try to modify its code to trigger the suspicious parts.

7.2.2 Long Term Perspectives

Our static analysis work presented in this dissertation is done on the bytecode available inside the .apk file at the analysis time. This method can miss any code that is downloaded dynamically from a remote server at runtime, it does not analyze it statically. Nevertheless, TriggerDroid is configured to mark dynamic code loading as suspicious, and it launches the part of the application that loads code. Any good monitoring tool can catch and analyze the code loaded at runtime. TriggerDroid is *deliberately* designed to not classify the executed code and let this task to other tools. It solely makes sure that the targeted code location was indeed triggered, in this case, the code that dynamically loads another code. However, TriggerDroid can be extended to catch and analyze the dynamically loaded code. Indeed, all the pieces that analyze the bytecode, modify it, launch the application and checks if the targeted code was launched are implemented in TriggerDroid. We believe that adding a module to analyze dynamically loaded code is feasible.

Similarly, CFGs are built only from the `.dex` bytecode. A possible extension of our work could be to analyze native and dynamically loaded code to complete the CFG. For instance, we could add a module that adjusts the CFG on the fly to add method CFGs and interprocedural edges when a bytecode class is dynamically loaded.

Malicious code protections have evolved over time since the apparition of the first Android malware. However, as far as we know, there is no large scale study that highlights evolution of these protection mechanisms. We believe that this part of the literature is not well studied and it merits further investigations, because it can help to design better analysis and detection solutions.

Nowadays, collecting a large scale dataset is not a hard problem. Nevertheless, giving sufficient information about its samples is indeed a non-trivial task. Since we had some experiment with malware dataset, we believe that a good one should include for each malware at least information about: the nature of the malicious actions. How are they triggered? Is the malware obfuscated? Is it piggybacked inside another app? What are the platform versions on which this malware can be executed? etc. This can

be done manually only for few samples because of the huge required effort. However, we believe that analysis techniques are quite mature to give a lot of information about malware. This kind of dataset could be existing but, as far as we know, there is no public one. For instance, Google is known for having rich reports about Android applications. This includes static and dynamic features, and developer relationships to detect non-harmful applications that are created by a developer that may have been previously associated with the creation of a malware [102].

Author's Publications

These are the publications of the author during his PhD thesis:

1. Mourad Leslous, Valérie Viet Triem Tong, Jean-François Lalande, Thomas Genet. GPFinder: Tracking the Invisible in Android Malware. 12th International Conference on Malicious and Unwanted Software, Oct 2017, Fajardo, Puerto Rico. IEEE Conputer Society, 12th International Conference on Malicious and Unwanted Software, 2017.

2. Mourad Leslous, Jean-François Lalande, and Valérie Viet Triem Tong. "Poster: Using Implicit Calls to Improve Malware Dynamic Execution." 37th IEEE Symposium on Security and Privacy. 2016. Publisher: IEEE Computer Society.

3. Valérie Viet Triem Tong, Aurélien Trulla, Mourad Leslous, Jean-François Lalande. Information flows at OS level unmask sophisticated Android malware. 14th International Conference on Security and Cryptography, Jul 2017, Madrid, Spain. SciTePress, 6, pp.578-585, 2017

4. Nicolas Kiss and Jean-Francois Lalande and Mourad Leslous and Valérie Viet Triem Tong. Kharon Dataset: Android Malware under a Microscope. USENIX Association. The LASER Workshop: Learning from Authoritative Security Experiment Results. 2016 May 26:1.

5. Valérie Viet Triem Tong, Jean-François Lalande, Mourad Leslous. Challenges in Android Malware Analysis. ERCIM News, ERCIM, 2016, Special Theme: Cybersecurity, pp.42-43.

6. Jean-François Lalande, Valérie Viêt Triem Tong, Mourad Leslous, Pierre Graux. Challenges for Reliable and Large Scale Evaluation of Android Malware Analysis. SHPCS 2018 - International Workshop on Security and High Performance Computing Systems, Jul 2018, Orléans, France. IEEE Computer Society, pp.1-3

Bibliography

[1] IDC, *Smartphone os market share, 2016 q2*, http://www.idc.com/prodserv/smartphone-os-market-share.jsp, Accessed: 2016-10-21, 2016.

[2] Statista, *Number of available applications in the google play store from december 2009 to december 2017*, 2018.

[3] Google, *Android security : 2017 year in review*, https://source.android.com/security/reports/Google_Android_Security_2017_Re-port_Final.pdf, 2018.

[4] P. Faruki, A. Bharmal, V. Laxmi, V. Ganmoor, M. S. Gaur, M. Conti, and M. Rajarajan, "Android security: A survey of issues, malware penetration, and defenses", *IEEE Communications Surveys Tutorials*, vol. 17, no. 2, pp. 998–1022, 2015, ISSN: 1553-877X. DOI: 10.1109/COMST.2014.2386139.

[5] L. Li, D. Li, T. F. Bissyandé, J. Klein, Y. Le Traon, D. Lo, and L. Cavallaro, "Understanding android app piggybacking: A systematic study of malicious code grafting", *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 6, pp. 1269–1284, 2017.

[6] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps", in *35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14, Edinburgh, United Kingdom: ACM, 2014, pp. 259–269.

[7] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna, "Execute this! analyzing unsafe and malicious dynamic code loading in android applications.", in *Network and Distributed System Security (NDSS) Symposium*, vol. 14, 2014, pp. 23–26.

[8] A. Apvrille and R. Nigam, *Obfuscation in android malware, and how to fight back*, https://www.virusbulletin.com/virusbulletin/2014/07/obfus-cation-android-malware-and-how-fight-back, 2014.

[9] V. Rastogi, Y. Chen, and X. Jiang, "Droidchameleon: Evaluating android anti-malware against transformation attacks", in *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, ACM, 2013, pp. 329–334.

[10] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro, "Copperdroid: Automatic reconstruction of android malware behaviors.", in *NDSS*, 2015.

[11] Y. Fratantonio, A. Bianchi, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna, "Triggerscope: Towards detecting logic bombs in android applications", in *Security and Privacy (SP), 2016 IEEE Symposium on*, IEEE, 2016, pp. 377–396.

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

[13] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, "Crowdroid: Behavior-based malware detection system for android", in *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, ACM, 2011, pp. 15–26.

[14] H. Gunadi and A. Tiu, "Efficient runtime monitoring with metric temporal logic: A case study in the android operating system", in *International Symposium on Formal Methods*, Springer, 2014, pp. 296–311.

[15] F. E. Allen, "Control flow analysis", in *Proceedings of a Symposium on Compiler Optimization*, Urbana-Champaign, Illinois: ACM, 1970, pp. 1–19. DOI: 10.1145/800028.808479.

[16] X. Pan, X. Wang, Y. Duan, X Wang, and H. Yin, "Dark hazard: Learning-based, large-scale discovery of hidden sensitive operations in android apps", in *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)*, 2017.

[17] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan, "Puma: Programmable ui-automation for large-scale dynamic analysis of mobile apps", in *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*, ACM, 2014, pp. 204–217.

[18] S. Rasthofer, S. Arzt, S. Triller, and M. Pradel, "Making malory behave maliciously: Targeted fuzzing of android execution environments", in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE '17, Buenos Aires, Argentina: IEEE Press, 2017, pp. 300–311, ISBN: 978-1-5386-3868-2. DOI: 10.1109/ICSE.2017.35. [Online]. Available: \url{https://doi.org/10.1109/ICSE.2017.35}.

[19] A. Abraham, R. Andriatsimandefitra, A. Brunelat, J.-F. Lalande, and V. Viet Triem Tong, "Grodddroid: A gorilla for triggering malicious behaviors", in *10th International Conference on Malicious and Unwanted Software*, IEEE Computer Society, 2015.

[20] S. Rasthofer, S. Arzt, M. Miltenberger, and E. Bodden, "Harvesting runtime data in android applications for identifying malware and enhancing code analysis", Technical Report TUD-CS-2015-0031, EC SPRIDE, Tech. Rep., 2015.

[21] F. Wei, Y. Li, S. Roy, X. Ou, and W. Zhou, "Deep ground truth analysis of current android malware", in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, Springer, 2017, pp. 252–276.

[22] M. Leslous, V. Viet Triem Tong, J.-F. Lalande, and T. Genet, "GPFinder: Tracking the Invisible in Android Malware", in *12th International Conference on Malicious and Unwanted Software*, Fajardo, Puerto Rico: IEEE Conputer Society, Oct. 2017. [Online]. Available: https://hal-centralesupelec.archives-ouvertes.fr/hal-01584989.

[23] N. Kiss, J.-F. Lalande, M. Leslous, and V. V. T. Tong, "Kharon dataset: Android malware under a microscope", in *The LASER Workshop: Learning from Authoritative Security Experiment Results (LASER 2016)*, San Jose, CA: USENIX Association, 2016, pp. 1–12, ISBN: 978-1-931971-35-5. [Online]. Available: \url{https://www.usenix.org/conference/laser2016/program/presentation/kiss}.

[24] Z. Aung and W. Zaw, "Permission-based android malware detection", *International Journal of Scientific and Technology Research*, vol. 2, no. 3, pp. 228–234, 2013.

[25] H. Peng, C. Gates, B. Sarma, N. Li, Y. Qi, R. Potharaju, C. Nita-Rotaru, and I. Molloy, "Using probabilistic generative models for ranking risks of android apps", in *Proceedings of the 2012 ACM conference on Computer and communications security*, ACM, 2012, pp. 241–252.

[26] V. Moonsamy, J. Rong, and S. Liu, "Mining permission patterns for contrasting clean and malicious android applications", *Future Generation Computer Systems*, vol. 36, pp. 122–132, 2014.

[27] C.-Y. Huang, Y.-T. Tsai, and C.-H. Hsu, "Performance evaluation on permission-based detection for android malware", in *Advances in Intelligent Systems and Applications-Volume 2*, Springer, 2013, pp. 111–120.

[28] J. Schütte, R. Fedler, and D. Titze, "Condroid: Targeted dynamic analysis of android applications", in *2015 IEEE 29th International Conference on Advanced Information Networking and Applications*, 2015, pp. 571–578. DOI: 10.1109/AINA.2015.238.

[29] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou, "Smartdroid: An automatic system for revealing ui-based trigger conditions in android applications", in *2nd ACM workshop on Security and privacy in smartphones and mobile devices*, ACM, 2012, pp. 93–104.

[30] S. Smalley and R. Craig, "Security enhanced (se) android: Bringing flexible mac to android.", in *NDSS*, vol. 310, 2013, pp. 20–38.

[31] P. A. Loscocco, S. D. Smalley, P. A. Muckelbauer, R. C. Taylor, S. J. Turner, and J. F. Farrell, "The inevitability of failure: The flawed assumption of security in modern computing environments", in *Proceedings of the 21st National Information Systems Security Conference*, vol. 10, 1998, pp. 303–314.

[32] F. E. Allen, "Control flow analysis", in *Proceedings of a Symposium on Compiler Optimization*, Urbana-Champaign, Illinois: ACM, 1970, pp. 1–19. DOI: 10.1145/800028.808479. [Online]. Available: \url{http://doi.acm.org/10.1145/800028.808479}.

[33] E. J. Chikofsky and J. H. Cross, "Reverse engineering and design recovery: A taxonomy", *IEEE software*, vol. 7, no. 1, pp. 13–17, 1990.

[34] R. Vallee-Rai and L. J. Hendren, "Jimple: Simplifying java bytecode for analyses and transformations", 1998.

[35] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot - a java bytecode optimization framework", in *1999 Conference of the Centre for Advanced Studies on Collaborative Research*, Mississauga, Ontario, Canada: IBM Press, 1999, pp. 13–.

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

[37] J. Dean, D. Grove, and C. Chambers, "Optimization of object-oriented programs using static class hierarchy analysis", in *European Conference on Object-Oriented Programming*, Springer, 1995, pp. 77–101.

[38] D. F. Bacon and P. F. Sweeney, "Fast static analysis of c++ virtual function calls", *ACM Sigplan Notices*, vol. 31, no. 10, pp. 324–341, 1996.

[39] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, and C. Godin, *Practical virtual method call resolution for Java*, 10. ACM, 2000, vol. 35.

[40] O. Lhoták and L. Hendren, "Scaling java points-to analysis using spark", in *Compiler Construction*, G. Hedin, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 153–169, ISBN: 978-3-540-36579-2.

[41] L. O. Andersen, "Program analysis and specialization for the c programming language", PhD thesis, University of Cophenhagen, 1994.

[42] B. Steensgaard, "Points-to analysis in almost linear time", in *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ACM, 1996, pp. 32–41.

[43] X. Zhou, S. Demetriou, D. He, M. Naveed, X. Pan, X. Wang, C. A. Gunter, and K. Nahrstedt, "Identity, location, disease and more: Inferring your secrets from android public resources", in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, ACM, 2013, pp. 1017–1028.

[44] G. Meng, Y. Xue, C. Mahinthan, A. Narayanan, Y. Liu, J. Zhang, and T. Chen, "Mystique: Evolving android malware for auditing anti-malware tools", in *Proceedings of the 11th ACM on Asia conference on computer and communications security*, ACM, 2016, pp. 365–376.

[45] Y. Zhang, Y. Li, T. Tan, and J. Xue, "Ripple: Reflection analysis for android apps in incomplete information environments", *Software: Practice and Experience*, vol. 48, no. 8, pp. 1419–1437, 2018.

[46] L. Li, T. F. Bissyandé, D. Octeau, and J. Klein, "Droidra: Taming reflection to support whole-program analysis of android apps", in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ACM, 2016, pp. 318–329.

[47] G. Suarez-Tangil, J. E. Tapiador, F. Lombardi, and R. Di Pietro, "Alterdroid: Differential fault analysis of obfuscated smartphone malware", *IEEE Transactions on Mobile Computing*, vol. 15, no. 4, pp. 789–802, 2016.

[48] Y. Chen, Y. Zhang, Z. Wang, L. Xia, C. Bao, and T. Wei, "Adaptive android kernel live patching", in *Proceedings of the 26th USENIX Security Symposium (USENIX Security 17)*, 2017.

[49] C. Ren, P. Liu, and S. Zhu, "Windowguard: Systematic protection of gui security in android", in *Proc. of the Annual Symposium on Network and Distributed System Security (NDSS)*, 2017.

[50] G. Costa, P. Gasti, A. Merlo, and S.-H. Yu, "Flex: A flexible code authentication framework for delegating mobile app customization", in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ACM, 2016, pp. 389–400.

[51] J. Seo, D. Kim, D. Cho, I. Shin, and T. Kim, "Flexdroid: Enforcing in-app privilege separation in android.", in *NDSS*, 2016.

[52] M. Backes, S. Bugiel, C. Hammer, O. Schranz, and P. v. Styp-Rekowsky, "Boxify: Full-fledged app sandboxing for stock android", 2015.

[53] X. Wang, K. Sun, Y. Wang, and J. Jing, "Deepdroid: Dynamically enforcing enterprise policy on android devices.", in *NDSS*, 2015.

[54] G. S. Tuncay, S. Demetriou, K. Ganju, and C. A. Gunter, "Resolving the predicament of android custom permissions", in *ISOC Network and Distributed Systems Security Symposium (NDSS)*, 2018.

[55] S. Rasthofer, S. Arzt, E. Lovat, and E. Bodden, "Droidforce: Enforcing complex, data-centric, system-wide policies in android", in *Availability, Reliability and Security (ARES), 2014 Ninth International Conference on*, 2014, pp. 40–49.

[56] C. Yagemann and W. Du, "Intentio ex machina: Android intent access control via an extensible application hook", in *European Symposium on Research in Computer Security*, Springer, 2016, pp. 383–400.

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

[58] H. Fu, Z. Zheng, S. Bose, M. Bishop, and P. Mohapatra, "Leaksemantic: Identifying abnormal sensitive network transmissions in mobile applications", in *INFOCOM 2017-IEEE Conference on Computer Communications, IEEE*, IEEE, 2017, pp. 1–9.

[59] A. Continella, Y. Fratantonio, M. Lindorfer, A. Puccetti, A. Zand, C. Kruegel, and G. Vigna, "Obfuscation-resilient privacy leak detection for mobile apps through differential analysis", in *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)*, 2017, pp. 1–16.

[60] K. Xu, Y. Li, and R. H. Deng, "Iccdetector: Icc-based malware detection on android", *IEEE Transactions on Information Forensics and Security*, vol. 11, no. 6, pp. 1252–1264, 2016.

[61] R. Slavin, X. Wang, M. B. Hosseini, J. Hester, R. Krishnan, J. Bhatia, T. D. Breaux, and J. Niu, "Toward a framework for detecting privacy policy violations in android application code", in *Proceedings of the 38th International Conference on Software Engineering*, ACM, 2016, pp. 25–36.

[62] X. Chen and S. Zhu, "Droidjust: Automated functionality-aware privacy leakage analysis for android applications", in *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, ACM, 2015, p. 5.

[63] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, "Iccta: Detecting inter-component privacy leaks in android apps", in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, IEEE Press, 2015, pp. 280–291.

[64] V. Avdiienko, K. Kuznetsov, A. Gorla, A. Zeller, S. Arzt, S. Rasthofer, and E. Bodden, "Mining apps for abnormal usage of sensitive data", in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, IEEE Press, 2015, pp. 426–436.

[65] S. Arzt, S. Rasthofer, and E. Bodden, "Susi: A tool for the fully automated classification and categorization of android sources and sinks", *University of Darmstadt, Tech. Rep. TUDCS-2013-0114*, 2013.

[66] A. Martín, A. Calleja, H. D. Menéndez, J. Tapiador, and D. Camacho, "Adroit: Android malware detection using meta-information", in *Computational Intelligence (SSCI), 2016 IEEE Symposium Series on*, IEEE, 2016, pp. 1–8.

[67] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, "Riskranker: Scalable and accurate zero-day android malware detection", in *Proceedings of the 10th international conference on Mobile systems, applications, and services*, ACM, 2012, pp. 281–294.

[68] G. Suarez-Tangil, J. E. Tapiador, P. Peris-Lopez, and J. Blasco, "Dendroid: A text mining approach to analyzing and classifying code structures in android malware families", *Expert Systems with Applications*, vol. 41, no. 4, pp. 1104–1117, 2014.

[69] G. Suarez-Tangil, S. K. Dash, M. Ahmadi, J. Kinder, G. Giacinto, and L. Cavallaro, "Droidsieve: Fast and accurate classification of obfuscated android malware", in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, ACM, 2017, pp. 309–320.

[70] S. Mutti, Y. Fratantonio, A. Bianchi, L. Invernizzi, J. Corbetta, D. Kirat, C. Kruegel, and G. Vigna, "Baredroid: Large-scale analysis of android apps on real devices", in *Proceedings of the 31st Annual Computer Security Applications Conference*, ACM, 2015, pp. 71–80.

[71] F. Maggi, A. Valdi, and S. Zanero, "Andrototal: A flexible, scalable toolbox and service for testing mobile malware detectors", in *Proceedings of the Third ACM workshop on Security and privacy in smartphones & mobile devices*, ACM, 2013, pp. 49–54.

[72] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou, "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*, ACM, 2012, pp. 93–104.

[73] S. K. Dash, G. Suarez-Tangil, S. Khan, K. Tam, M. Ahmadi, J. Kinder, and L. Cavallaro, "Droidscribe: Classifying android malware based on runtime behavior", in *Security and Privacy Workshops (SPW), 2016 IEEE*, IEEE, 2016, pp. 252–261.

[74] R. Spreitzer, F. Kirchengast, D. Gruss, and S. Mangard, "Procharvester: Fully automated analysis of procfs side-channel leaks on android", in *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, ACM, 2018, pp. 749–763.

[75] L. Li, D. Li, T. F. Bissyandé, J. Klein, H. Cai, D. Lo, and Y. L. Traon, "Automatically locating malicious packages in piggybacked android apps", in *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems*, IEEE Press, 2017, pp. 170–174.

[76] L.-K. Yan and H. Yin, "Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis.", in *USENIX security symposium*, 2012, pp. 569–584.

[77] F. Bellard, "Qemu, a fast and portable dynamic translator.", in *USENIX Annual Technical Conference, FREENIX Track*, vol. 41, 2005, p. 46.

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

[79] Y. Aafer, W. Du, and H. Yin, "DroidAPIMiner: Mining api-level features for robust malware detection in android", in *Security and Privacy in Communication Networks*, ser. LNICST, vol. 127, Sydney, NSW, Australia: Springer International Publishing, 2013, pp. 86–103.

[80] X. Hu, T.-c. Chiueh, and K. G. Shin, "Large-scale malware indexing using function-call graphs", in *Proceedings of the 16th ACM conference on Computer and communications security*, ACM, 2009, pp. 611–620.

[81] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X.-y. Zhou, and X. Wang, "Effective and efficient malware detection at the end host.", in *USENIX security symposium*, vol. 4, 2009, pp. 351–366.

[82] H. Gascon, F. Yamaguchi, D. Arp, and K. Rieck, "Structural detection of android malware using embedded call graphs", in *Proceedings of the 2013 ACM workshop on Artificial intelligence and security*, ACM, 2013, pp. 45–54.

[83] M. Lillack, C. Kästner, and E. Bodden, "Tracking load-time configuration options", in *29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '14, Vasteras, Sweden: ACM, 2014, pp. 445–456, ISBN: 978-1-4503-3013-8. DOI: 10.1145/2642937.2643001.

[84] W. Klieber, L. Flynn, A. Bhosale, L. Jia, and L. Bauer, "Android taint flow analysis for app sets", in *3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*, ser. SOAP '14, Edinburgh, United Kingdom: ACM, 2014, pp. 1–6, ISBN: 978-1-4503-2919-4. DOI: 10.1145/2614628.2614633.

[85] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon, "Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis", in *22Nd USENIX Conference on Security*, ser. SEC'13, Washington, DC, USA: USENIX Association, 2013, pp. 543–558, ISBN: 978-1-931971-03-4.

[86] M. Graa, N. Cuppens-Boulahia, F. Cuppens, and A. Cavalli, "Detecting control flow in smarphones: Combining static and dynamic analyses", in *4th International Symposium on Cyberspace Safety and Security*, Melbourne, Australia: Springer Berlin Heidelberg, 2012, pp. 33–47, ISBN: 978-3-642-35362-8. DOI: 10.1007/978-3-642-35362-8_4.

[87] T. Wu, J. Liu, Z. Xu, C. Guo, Y. Zhang, J. Yan, and J. Zhang, "Light-weight, inter-procedural and callback-aware resource leak detection for android apps", *IEEE Transactions on Software Engineering*, vol. 42, no. 11, pp. 1054–1076, 2016, ISSN: 0098-5589. DOI: 10.1109/TSE.2016.2547385.

[88] M. Junaid, D. Liu, and D. Kung, "Dexteroid: Detecting malicious behaviors in android apps using reverse-engineered life cycle models", *Computers and Security*, vol. 59, pp. 92 –117, 2016, ISSN: 0167-4048. DOI: http://doi.org/10.1016/j.cose.2016.01.008.

[89] S. Yang, D. Yan, H. Wu, Y. Wang, and A. Rountev, "Static control-flow analysis of user-driven callbacks in android applications", in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, IEEE Press, 2015, pp. 89–99.

[90] A. Salem, "Stimulation and detection of android repackaged malware with active learning", *arXiv preprint arXiv:1808.01186*, 2018.

[91] Y. Cao, Y. Fratantonio, A. Bianchi, M. Egele, C. Kruegel, G. Vigna, and Y. Chen, "EdgeMiner: Automatically detecting implicit control flow transitions through the android framework", in *The 2015 Network and Distributed System Security*, San Diego, CA, USA, 2015.

[92] L. Li, D. Li, T. F. Bissyandé, J. Klein, Y. Le Traon, D. Lo, and L. Cavallaro, "Understanding android app piggybacking: A systematic study of malicious code grafting", *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 6, pp. 1269–1284, 2017.

[93] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Androzoo: Collecting millions of android apps for the research community", in *13th International Conference on Mining Software Repositories*, ACM, 2016, pp. 468–471.

[94] R. Vallee-rai and L. Hendren, "Jimple: Simplifying java bytecode for analyses and transformations", Jan. 2004.

[95] L. de Moura and N. Bjørner, "Z3: An efficient smt solver", in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340, ISBN: 978-3-540-78800-3.

[96] A. Mohaisen and O. Alrawi, "Av-meter: An evaluation of antivirus scans and labels", in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, Springer, 2014, pp. 112–131.

[97] F. Maggi, A. Bellini, G. Salvaneschi, and S. Zanero, "Finding non-trivial malware naming inconsistencies", in *International Conference on Information Systems Security*, Springer, 2011, pp. 144–159.

[98] M. Hurier, G. Suarez-Tangil, S. K. Dash, T. F. Bissyandé, Y. L. Traon, J. Klein, and L. Cavallaro, "Euphony: Harmonious unification of cacophonous anti-virus vendor labels for android malware", in *Proceedings of the 14th International Conference on Mining Software Repositories*, IEEE Press, 2017, pp. 425–435.

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

[100] A. Salem and A. Pretschner, "Poking the bear: Lessons learned from probing three android malware datasets", in *Proceedings of the 1st International Workshop on Advances in Mobile App Analysis*, ACM, 2018, pp. 19–24.

[101] L. Georget, M. Jaume, F. Tronel, G. Piolle, and V. V. T. Tong, "Verifying the reliability of operating system-level information flow control systems in linux", in *Formal Methods in Software Engineering (FormaliSE), 2017 IEEE/ACM 5th International FME Workshop on*, IEEE, 2017, pp. 10–16.

[102] Google, *Android security : 2016 year in review*, `https://source.android.com/security/reports/Google_Andro-id_Security_2016_Report_Final.pdf`, 2017.

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE RENNES 1

Titre : Mettre en avant et exécuter les chemins suspicieux dans les malwares Android

Mots clés : Android, malware, analyse, CFG, condition de déclenchement

Résumé : Android est installé sur plus de 80% des smartphones aujourd'hui. Ses applications recueillent beaucoup d'informations sur l'utilisateur. Par conséquent, il est devenu une cible favorite des cybercriminels. Des approches récentes tentent de caractériser automatiquement le comportement malveillant en forçant l'exécution du code suspicieux. Elles se basent sur les CFG. Néanmoins, Ces CFG ne sont pas complets car ils ne prennent pas en considération les chemins d'exécution qui passent par le framework. Nous proposons dans ce mémoire un outil, GPFinder, qui extrait automatiquement les chemins d'exécution qui mènent vers les endroits suspicieux du code. Il fournit aussi des informations clés sur les applications analysées. Nous utilisons GPFinder pour étudier une collection de 14224 échantillons de malwares. Les approches de déclenchement de malware actuelles soient elles modifient excessivement l'application et perdent son contexte, soient elles manipulent les entrées de l'application sans modifier son code. Cependant, toutes les conditions de déclenchement ne sont pas faciles à contourner en manipulant seulement les entrées. Nous proposons dans ce mémoire un outil, TriggerDroid, qui a pour objectif de forcer l'exécution du code suspicieux et garder son contexte proche de l'originel. Il génère les événements framework requis pour lancer le bon composant de l'application, et il satisfait les conditions de déclenchement nécessaires pour prendre les chemins d'exécution voulus. Afin de valider notre approche, nous menons une expérience sur un ensemble de 135 échantillons de malware de 71 familles différentes.

Title : Highlight and Execute Suspicious Paths in Android Malware

Keywords : Android, malware, analysis, CFG, triggering condition

Abstract : Android is installed on more than 80% of today's smartphones. Its apps collect a huge amount of user data. Consequently, it has become a favorable target for cyber criminals. Thus, recent approaches try to automatically characterize the malicious behavior by forcing the execution of the suspicious code. They strongly rely on CFGs. However, these CFGs are incomplete because they do not take into consideration execution paths that pass through the framework. We propose in this dissertation a tool, GPFinder, that automatically exhibits execution paths towards suspicious locations and gives key information about the analyzed applications. We use GPFinder to study a collection of 14,224 malware samples. Current malware triggering approaches either heavily modify the app and lose its context, or fuzz the input without modifying the app's code. However, not all triggering conditions can be bypassed solely by fuzzing the input. We propose in this dissertation a tool, TriggerDroid, that has a twofold goal: force the execution of the suspicious code, and keep its context close to the original one. It crafts the required framework events to launch the right app component, and satisfies the necessary triggering conditions to take the desired execution path. To validate our approach, we led an experiment on a dataset of 135 malware samples from 71 different families.