University of Piraeus
School of Information and Communication Technologies
Department of Digital Systems

Security and Privacy Enhancing Mechanisms for The Android Operating System

Doctoral Thesis

Christos Lyvas

Piraues, December 2021

University of Piraeus
School of Information and Communication Technologies
Department of Digital Systems

Security and Privacy Enhancing Mechanisms for The Android Operating System

Supervisor: Prof. Costas Lambrinoudakis

Advisors: Prof. Christos Xenakis
Asst. Prof. Panagiotis Rizomiliotis

This thesis is submitted for the degree of
Doctor of Philosophy

Piraues, December 2021

APPROVAL SHEET
UNIVERSITY OF PIRAEUS
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGIES
DEPARTMENT OF DIGITAL SYSTEMS

This is to certify that the Thesis presented by Christos Lyvas, entitled **"Security and Privacy Enhancing Mechanisms for The Android Operating System"**, submitted in fulfillment of the requirement for the degree of Doctor of Philosophy, complies with the regulation of the University of Piraeus and meets the accepted standards with respect to originality.

Costas Lambrinoudakis Professor University of Piraeus	[illegible]	17 December 2021
(Supervisor)	(Signature)	(Date)
Christos Xenakis Professor University of Piraeus	[illegible]	17 December 2021
(Advisor)	(Signature)	(Date)
Panagiotis Rizomiliotis Assistant Professor Harokopio University of Athens	[illegible]	17 December 2021
(Advisor)	(Signature)	(Date)
Stefanos Gritzalis Professor University of Piraeus	[illegible]	17 December 2021
(External Examiner)	(Signature)	(Date)

Sokratis Katsikas Professor Norwegian University of Science and Technology	[illegible]	17 December 2021
(External Examiner)	(Signature)	(Date)
Christos Kalloniatis Associate Professor University of Aegean	[illegible]	17 December 2021
(External Examiner)	(Signature)	(Date)
Nikolaos Pitropakis Associate Professor Edinburgh Napier University	[illegible]	17 December 2021
(External Examiner)	(Signature)	(Date)

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this thesis are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This thesis is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements. I additionally declare that the opinions expressed in this document are my sole responsibility and do not necessarily represent the official position of the University of Piraeus.

[illegible]

Christos Lyvas
Piraeus, December 7, 2021

To my grandfather, who used to ask me every day if I got my Ph.D.,
as well as my parents and Chrys for their tolerance and support.

"People think of education as something that they can finish."

Isaac Asimov

Acknowledgements

There are many people I would like to express my appreciation and gratitude for their contributions to this dissertation and my scientific evolvement over the past years. First, I would like to express my sincere gratitude to my supervisor Professor Costas Lambrinoudakis for his kindness, invaluable mentoring, endless support, guidance, and confidence in me and my potential.

This dissertation would also not have been possible without the support, excellent cooperation, and contribution of Dr. Dimitris Geneiatakis, which was very constructive during my studies.

I would like to thank the advisory committee, Professor Christos Xenakis, and Assistant Professor Panagiotis Rizomiliotis.

I would like to extend my appreciation to Professor Athanasios G. Kanatas and Professor Stefanos Gritzalis for their faith in me and for allowing me to participate in several national and European projects that broadened my horizons to several scientific fields.

Moreover, I would like to thank Associate Professor Christos Kalloniatis and Assistant Professor Konstantinos Maliatsos for their excellent cooperation and guidance during my participation in research projects in recent years.

Finally, I would like to express my appreciation to the Systems Security Laboratory and Telecommunication Systems Laboratory members of the University of Piraeus and particularly Dr. Viktor Nikolaidis for their support and friendship.

Christos Lyvas
Piraeus, December 7, 2021

Abstract

The Android platform is the dominant Operating System (OS) for mobile and IoT devices. Its wide distribution is mainly due to the freedom it grants to mobile manufacturers (OEMs) to use it as the primary operating system for their devices. In terms of security, Android's enhanced security model safeguards the end-users from the threats of conventional Operating Systems (macOS, Windows, Linux, *etc.*). However, Android has vulnerabilities due to its architecture particularities. This thesis focuses on the threats, vulnerabilities and security mechanisms of the Android Operating System (OS).

Initially, the permission-based access control model of the Android Operating System for controlling access to sensitive phone resources and its "relation' to the Android framework's methods for constructing the permission maps has been investigated. The aforementioned "relation" between a framework method and a permission can be found in the Android documentation. However, in addition to the fact that documentation may accidentally lack information, Android features undocumented and hidden API methods. To this direction, this thesis proposes *Dypermin*, a transparent framework for compiling the Android permission map without requiring any modification to the underlying operating system. To achieve that, *Dypermin* capitalizes on intrinsic properties of the Android framework, that is, security exceptions during run time and the availability of any protected API method through the Android framework, as well as on the advantages of the Java reflection mechanism. Furthermore, *Dypermin*, in contrast to other related methods, validates itself as it relies on run time information, meaning that it does not generate false-positive map entries. *Dypermin* has been evaluated on different Android versions. The results have been compared with those of other proposed methods demonstrating *Dypermin's* efficacy for compiling the Android permission map for any given version.

Furthermore, it has been demonstrated that the Android's activity and task hijacking attacks may have a significant impact on end users' data confidentiality since malicious applications can deceive end-users and silently gain access to sensitive data. Undoubtedly, such attacks are of vital importance and thus, for their thorough study, a tool named *Anactijax* has been proposed, used for identifying specific configurations that an application may be vulnerable to. Furthermore, an operating system level defense mechanism named *TaskAuth* has been proposed, controlling the access to the applications' activities. *TaskAuth* is transparent to both end-users and developers by leveraging their built-in signatures. The effectiveness of *TaskAuth* has been evaluated against various vulnerable configurations provided by *Anactijax*. Results have proved that the proposed solution does not affect Android's task management and that end-users suffer negligible execution overhead.

Finally, the Android intent redirection and intent hijacking attacks were examined. Again this type of attacks can be launched by malicious applications in order to gain access to users' sensitive data. For addressing this type of attacks an operating system-level defense mechanism named *IntentAuth* has been proposed. *IntentAuth* allows end-users to explicitly define dynamic policies that specify a trust model for applications allowed to interact with each other. Additionally, upon the user-defined trust model for installed applications, the proposed implementation can verify and encrypt the data transmitted (application to application) during the Android's Inter-Process communication mechanism via implicit Intents. The execution overhead imposed has been proved to be negligible.

Keywords: Android Security, Activity Hijacking Attacks, Task Hijacking Attacks, Task Hijacking Prevention, Activity Hijacking Prevention, Mobile Operating System Security, Mobile Application Security, Mobile Application Privacy, Android Privacy, Intent Redirection Attacks Prevention, Intent Hijacking Prevention, Intent Encryption, Intent Authorization, Intent Authentication, Android Permissions Map, Reflection Technique, Dynamic Analysis

Περίληψη

Η πλατφόρμα Android αποτελεί το κυρίαρχο λειτουργικό σύστημα για κινητές και IoT συσκευές. Η ευρεία διάδοση του οφείλεται στην ελευθερία χρήσης που δίνετε στους κατασκευαστές κινητών συσκευών (Original Equipment Manufacturers - OEMs) να το χρησιμοποιούν ως το κύριο λειτουργικό σύστημα για τις συσκευές τους. Αφενός μεν, το μοντέλο ασφαλείας του Android προστατεύει τους χρήστες από τις απειλές των συμβατικών λειτουργικών συστημάτων όπως macOS, Windows, Linux, κλπ. Αφετέρου δε, στο Android εμφανίζονται αδυναμίες εξαιτίας των ιδιαιτεροτήτων της αρχιτεκτονικής του. Η παρούσα διδακτορική διατριβή επικεντρώνεται στο λειτουργικό σύστημα Android, τους μηχανισμούς ασφάλειας του, τις απειλές του, τα τρωτά σημεία του, και την αποτροπή σειράς επιθέσεων σε αυτό.

Αρχικά διερευνάτε το μοντέλο ελέγχου αδειών (permission model) του λειτουργικού συστήματος για τον περιορισμό της πρόσβασης σε ευαίσθητους πόρους από εφαρμογές και τη "σχέση" του με τις μεθόδους API (Application Programming Interface) του Android framework για την κατασκευή των συσχετίσεων αδειών (permissions) και των μεθόδων του framework. Η συσχέτιση μεταξύ μιας μεθόδου API του Android framework και ενός permission μπορεί να βρεθεί μέσω του documentation των μεθόδων του συστήματος Android. Ωστόσο, όχι μόνο το documentation μπορεί τυχαία να στερείται πληροφοριών αλλά και το Android διαθέτει undocumented και private μεθόδους API. Το αποτέλεσμα της ανάλυσης ήταν το *Dypermin*, ένα εργαλείο ικανό να συσχετίσει μεθόδους API και permissions χωρίς να απαιτείται τροποποίηση του υποκείμενου λειτουργικού συστήματος. Για να επιτευχθεί αυτό, το *Dypermin* αξιοποιεί τις εξαιρέσεις ασφαλείας (runtime Java security exceprtions) κατά τη διάρκεια εκτέλεσης μεθόδων του Android Framework και τη τεχνική reflection της γλώσσας προγραμματισμού Java για την προσπέλαση οποιασδήποτε προστατευμένης (private) μεθόδου API του Android Framework. Επιπλέον, το *Dypermin*, σε αντίθεση με άλλες σχετικές μεθόδους, βασίζεται σε πληροφορίες κατά τους χρόνους εκτέλεσης των μεθόδων API του Android framework, πράγμα που σημαίνει ότι δεν δημιουργεί ψευδώς θετικές (false positives) συσχετίσεις. Το *Dypermin* αξιολογήθηκε σε διαφορετικές εκδόσεις Android και τα αποτελέσματα του συγκρίθηκαν με αντίστοιχα αποτελέσματα άλλων προτεινόμενων μεθόδων για την ανάδειξη της αποτελεσματικότητα του για οποιαδήποτε έκδοση του συστήματος.

Παράλληλα εξετάζονται οι επιθέσεις activity, και task hijacking στο λειτουργικό σύστημα Android που μπορούν να έχουν μεγάλο αντίκτυπο για τους χρηστες, καθώς κακόβουλες εφαρμογές που εκμεταλλεύονται τέτοιες αδυναμίες μπορούν να τους εξαπατήσουν και να αποκτήσουν πρόσβαση σε ευαίσθητα δεδομένα τους και λογαριασμούς τους. Αναμφίβολα, αυτές οι απειλές έχουν μεγάλη σημασία και για τη διεξοδική μελέτη τους, δημιουργήθηκε το εργαλείο *Anactijax*, ικανό να εντοπίσει συγκεκριμένες διαμορφώσεις στις οποίες μπορεί να είναι ευάλωτη μια εφαρμογή. Επιπλέον, προτάθηκε ένας μηχανισμός αποτροπής σε επίπεδο λειτουργικού συστήματος με όνομα *TaskAuth* που ελέγχει και περιορίζει την πρόσβαση των εφαρμογών σε activities. Το *TaskAuth* λειτουργεί με διαφάνεια τόσο για τους χρήστες όσο και για τους προγραμματιστές καθώς αξιοποιεί τις ενσωματωμένες υπογραφές που φέρουν οι εφαρμογές Android. Η αποτελεσματικότητα του *TaskAuth* αξιολογήθηκε σε σχέση με διάφορες ευάλωτες διαμορφώσεις που ανέδειξε to *Anactijax* και τα αποτελέσματα έδειξαν ότι η εφαρμογή της προτεινόμενης λύσης δεν επηρεάζει τη διαχείριση των tasks του Android ούτε επιβαρύνει τους χρόνους εκτέλεσης των εφαρμογών.

Τέλος, εξετάζονται οι απειλές intent redirection and intent hijacking στο σύστημα Android. Αυτές οι απειλές μπορούν να επηρεάσουν την εμπιστευτικότητα και την ακεραιότητα των δεδομένων των χρηστών, καθώς κακόβουλες εφαρμογές που εκμεταλλεύονται τέτοιες ευπάθειες σε άλλες εφαρμογές μπορούν να εξαπατήσουν τους χρήστες και να αποκτήσουν πρόσβαση ή να παραποιήσουν τα δεδομένα τους. Έτσι, μελετήθηκαν διεξοδικά οι απειλές intent redirection and intent hijacking και για την αποτροπή τους προτάθηκε ένας μηχανισμός προστασίας των χρηστών σε επίπεδο λειτουργικού συστήματος με όνομα *IntentAuth* που δίνει τη δυνατότητα στους χρήστες να καθορίζουν δυναμικές πολιτικές στα σύνολα των εφαρμογών που επιτρέπουν να αλληλοεπιδρούν μεταξύ τους. Επιπλέον, βάσει του μοντέλου εμπιστοσύνης που ο κάθε χρήστης καθορίζει για τις εγκατεστημένες εφαρμογές του, ο μηχανισμός *IntentAuth* μπορεί να κρυπτογραφήσει τα μεταδιδόμενα δεδομένα της διαδιεργασιακής επικοινωνίας του συστήματος μεταξύ εφαρμογών με implicit intents χωρίς επιβάρυνση του χρόνου εκτέλεσης.

Λέξεις Κλειδιά: Ασφάλεια Android, Επιθέσεις Activity Hijacking, Επιθέσεις Task Hijacking, Πρόληψη Επιθέσεων Activity Hijacking, Πρόληψη Επιθέσεων Task Hijacking, Ασφάλεια Λειτουργικών Συστημάτων, Ασφάλεια Εφαρμογών, Ιδιωτικότητα Εφαρμογών, Ιδιωτικότητα στο Android, Πρόληψη Επιθέσεων Intent Redirection, Πρόληψη Επιθέσεων Intent Hijacking, Κρυπτογραφία Διαδιεργασιακής Επικοινωνίας, Εξουσιοδότηση Διαδιεργασιακής Επικοινωνίας, Πιστοποίηση Δεδομένων Διαδιεργασιακής Επικοινωνίας, Android Permissions Map, Reflection Technique, Δυναμική Ανάλυση

Contents

List of Figures

List of Tables

List of Algorithms

Abbreviations and Acronyms

ADB	Android Debug Bridge
AOSP	Android Open Source Project
AOT	Ahead-Of-Time
API	Application Programming Interface
APK	AndroidPacKage
APP	Application
ART	Android Runtime Environment
ASLR	Address Space Layout Randomization
ATMS	Activity Task Manager Service
CA	Certificate Authority
CFG	Control-Flow Graph
CPU	Central Processing Unit
DAC	Discretionary Access Control
DEX	Dalvik Executable
DVM	Dalvik Virtual Machine
FBE	File Based Encryption
FDE	Full Disk Encryption

HAL	Hardware Abstraction Layer
HSM	Hardware Security Module
ICC	Inter-Component Communication
IMEI	International Mobile Equipment Identity
iOS	iPhone Operating System
IoT	Internet-of-Things
IPC	Inter-Process Communication
ITS	Intelligent Transportation Systems
JIT	Just-in-Time
JNI	Java Native Interface
JVM	Java Virtual Machine
MAC	Mandatory Access Control
NDK	Native Development Kit
NX	No eXecute
OEM	Original Equipment Manufacturers
OHA	Open Handset Alliance
OS	Operating System
PKI	Public Key Infrastructure
RAM	Random-Access Memory
RPC	Remote Procedure Calls
SDK	Software Development Kit
SMS	Short Message Service
SoC	System on Chip
TEE	Trusted Execution Environment

UI	User-Interface
UID	User Identifier
URI	Uniform Resource Identifier
VPN	Virtual Private Network
XML	eXtensible Markup Language

Chapter 1

Introduction

Nowadays, mobile phones are not only used for making phone calls but they also support our digital interactions with different types of services *i.e.*, banking, traveling, *etc.*, through the corresponding mobile applications (apps). In the mobile world, currently, Android is considered the dominant operating system (OS)[1], with more than three million available applications for downloading (only) from Google play[2]. In fact, as Android end users basis is constantly expanding, more and more malicious software (malware) targets their devices for fun and profit[3,4].

One pillar of Android's security is the process isolation at the kernel level, so that malevolent applications and services do not to affect the reliability of other services/applications or even of the device itself. Furthermore, it introduces an access control model, at application layer, for restricting access to "sensitive" resources (camera, location data, network, to mention a few) that could affect user's privacy or cause a security incident. Specifically, access to any sensitive resource is granted through a protected Application Programming Interface (API) method. An application in order to use a protected API method it must first declare the corresponding permissions in its manifest file, and request it also at runtime if it is executed on Android latest versions, otherwise a security exception is raised. In any case, users should give their consent for the permissions requested by the application either during the first time that a protected API method is invoked or during installation process, depending on the Android version. A more detailed analysis of the Android security model and the evolution of its permissions subsystem can be found in [38, 100].

[1]https://www.idc.com/promo/smartphone-market-share
[2]https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/
[3]https://www.mcafee.com/enterprise/en-us/assets/reports/rp-mobile-threat-report-2019.pdf
[4]https://www.mcafee.com/content/dam/consumer/en-us/docs/2020-Mobile-Threat-Report.pdf

Programmers get information on the correlation between permissions and protected API methods through Android's documentation. To this direction, an issue that attracts the attention of researchers, developers and Android enthusiasts is the question *"What is the exact correlation between Android Software Development Kit (SDK) API methods and permissions?"*. This is caused by the fact that (a) documentation may accidentally lack information, and (b) Android has hidden and internal API methods that are not directly accessible at the application layer since they are not included in the Development SDK. Though the latter cannot be directly accessed there are several publicly available sources[5] that give guidelines on how to gain access to such resources [58]. So eventually, programmers can gain access to these hidden API methods.

At this point it should be stressed that an accurate correlation between permissions and API methods is of high importance, as this correlation is utilized for malware detection [23] and other misconfigurations (*i.e.*, over-privileges [44]) identification. Today there are attempts, such as Stowaway [41], PScout [24], Axplorer [25] and Bartel *et al.* [26], to compile the correlation of API methods - permissions that extend Android's documentation. However, these approaches are bounded to specific Android versions and also require access to the underlying OS source code.

Moreover, as the Android OS evolves and in order to improve end-users' experiences, it proceeds with various modifications to the underlying subsystems. For instance, as already mentioned, permissions are enabled dynamically on the latest versions of Android, while from version 6.0 backwards they were granted statically. In addition, some API methods are deprecated, while other are introduced to support additional functionalities. So it is evident that these types of changes not only affect the API methods and permission mapping but also introduce inconsistencies in it among different Android versions according to [100]. Thus even other solutions, such as DPSpec [29], that exclusively rely on annotations (e.g., of documentation) cannot provide a complete coverage for the Android permission mapping.

The research work of this thesis [65] has elaborated on the developments of Android's API methods - permissions mapping, by proposing a framework, called *Dypermin*, capable of generating the permission map in a transparent way without requiring access to the OS source code and without generating false positive alarms. *Dypermin* automatically invokes Android's publicly accessible and hidden API methods in order to intentionally raise runtime security exceptions and thus decide whether or not a permission dependency exists.

More specifically, *Dypermin* relies on the simple observation that the Android OS raises a security exception if a protected API method is invoked without the appropriate permissions being defined in the application's manifest file. *Dypermin*, in order to identify and report the

[5] https://github.com/anggrayudi/android-hidden-api

permissions for every available API method, builds a single application that is automatically invoked after installation.

Dypermin achieves to extract all available API methods since it is provided through the SDK and thus it does not require any modifications to the underlying OS. Furthermore, it validates its finding as it relies on runtime information. *Dypermin* was evaluated with some well-known classes for different Android versions and its results are compared with both Android public and SDK source documentation as well as with the results of other related proposed methods. Currently, *Dypermin* does not provide a full mapping since it takes a substantial amount of time but it has been proved that it can accurately identify the API methods - permission map and deduce whether the available documentation is missing a relation between an API method and a permission.

Various types of malware [40], such as trojan, backdoors *etc.*, target mobile devices. Even though various protection techniques [54, 62, 72, 98] have been introduced in order to enhance their protection level and minimise the attack surface for Android platform, malicious entities do not only find new approaches to by pass the deployed countermeasures, but they also manage to launch their attacks silently [39, 97]. Researches have shown [84, 74] that attackers can bypass security checks and that malicious software can be published even in trusted sources.

In fact, some of Android security flaws are because of its divergent characteristics, something which is not the case for typical OS distributions. For instance, Android allows applications: (a) to monitor end users devices' sensors by granting the appropriate permission(s), (b) to interact directly with other applications, services and components through inter-process communication (IPC), and (c) combine activities from different applications into a certain task. Unlike other operating systems, an Android task is a combination of activities that end users interact with in order to perform a certain job; note that activities are the core element of Android's User Interfaces (UI) and are organized in tasks. Even though this functionality enables and improves end users experience on seamless transfer among different activities and applications, it may turn them vulnerable to activity injection [81, 89, 79, 80, 48] and task hijacking attacks [55, 48, 93, 61, 92]. Such attacks aim to deceive end users, by masquerading silently User Interfaces (UI), and steal credentials or other sensitive data provided to the targeted application.

Currently, various researchers have attempted to address task [55, 48, 93, 61, 92] and activity [81, 89, 79, 80] hijacking vulnerabilities, through various different approaches. While the effectiveness of these researches is not questionable, some of them have become obsolete [81, 89] due to Android framework's revisions, others require device 'rooting' [80] to enforce controls, others demand end users' intervention [93] assuming that their capable of recognizing malicious activities, others suffers from well-known limitations of static analysis in Android applications (for instance: disregard of implicit intents invocations) [55, 48]. Orthogonal solutions, such as [61], propose the deployment of additional static and dynamic analysis controls at vetting process, whereas authors in [92] introduce a system level mechanism, by modifying the underlying OS, in order to inform end users for possible activity hijacking attacks. In [48] a reformation of the current Android application development framework is required for introducing a signature based mechanism to protect Android applications against task hijacking attacks.

In an attempt to complement and improve the existing security provisions, research work presented in this thesis [66] aims to enhance the security of Android's task handling mechanism. The preconditions of task injection and activity hijacking attacks have been studied by introducing an open source tool called *Anactijax* that is capable of automatically generating activity injection test cases for assessing whether an application is vulnerable or not to such type of attacks. Moreover, they presented a system level authorization mechanism named *TaskAuth* that minimises the attack surface of Android's task handling mechanism by correlating the origin of interacting applications through their carrying signatures. *TaskAuth* controls the access among applications' activities using dynamic predefined policies *i.e.*, allow only vendor application or applications by the same developer to acquire a task from another application to their process. *TaskAuth* effectiveness has been evaluated in terms of possible implication on OS functionality and the overhead it introduces *i.e.*, how end user's experience is affected. The experiments performed highlight that *TaskAuth* does not affect Android's task handling core mechanism functionality, while the execution overhead introduced is considered negligible as it is not more than 4.3 ms in average.

Finally, another class of threats against the Android platform regards attacks against it's Inter-Process Communication (IPC) mechanism. As already mentioned, Android has divergent characteristics that are not found in typical OS distributions, such as the Inter-Process Communication (IPC) or, similarly, the Inter-Component Communication (ICC) mechanism similar to the "client-server" model. Due to its implementation peculiarities, Android's Inter-Process communication mechanism needs to employ protection measures against Intent Redirection [85, 37, 88] and Intent Hijacking [31, 95] attacks. The former category concerns cases where malicious applications leverage publicly available exported components (*i.e.*, Activities) and transfer data to other components that perform sensitive operations upon them. The latest threat concerns malicious applications that define Intent filters to intercept data transmitted by a legitimate application.

Currently, various research works attempt to address these security flaws using different approaches ranging from (a) static analysis detection tools [85, 37] to (b) system-level enhancements to prevent such attacks [88, 95] and minimize the OS's attack surface. While the effectiveness of these researches is not questionable, some of them suffer [85, 37] from the common limitations of static analysis approaches, while others that require the modification of the underlying OS, cannot be re-configured [88] dynamically, or expose intents beyond the interaction processes [95].

The research work of this thesis [68] complements and improves existing security provisions, by enhancing Android's IPC mechanism security. In particular, the security flaws associated with intent activity launch, intent hijacking and redirection have been thoroughly studied and a *novel and practical* open source system-level mechanism, namely *IntentAuth* (Intent Authentication), has been proposed enabling secure communication between applications and thus protecting users against such attacks. In fact, *IntentAuth's* goal is to minimize, if not eliminate, benign applications' attack surface by authenticating applications' communicating components through asymmetric key cryptography, and securing data exchange through symmetric key cryptography, employing user defined dynamic policies. In this way, users can control each applications' inter processes communication access transparently.

The rest of this thesis is structured as follows: Chapter 2 provides an overview and the main contributions of the conducted research. Chapter 3 provides an overview of the Android Operating System's core components, emphasizing on its security architecture. Chapter 4 briefly describes the Android Framework's permission model, the related threats and the methodologies proposed in the literature for the accurate construction of the Android's permission map. Chapter 5 describes the platform's task handling mechanism, the methodologies proposed in the literature for the investigation of the attack surface, and the mitigation mechanisms for eliminating the identified vulnerabilities. Chapter 6 presents the Android's Inter-Process Communication model with intents, its attack surface, and the mitigation mechanisms, as resulted from the relevant literature review, for addressing the identified threats. Finally, Chapter 7 concludes the thesis and provides remarks for future work.

Chapter 2

Contribution

Android is a custom Linux-based operating system. The primary running environment is the Dalvik Virtual Machine (DVM) similar to the Java Virtual Machine (JVM) in the initial versions or the Android Runtime Environment (ART) in the newer versions. Due to its architectural uniqueness, the Android Operating System (OS) is facing new types of threats and attacks. These new threats and attacks mainly concern the Android framework which adopts a "Client-Server" model for application interaction (Inter-Process Communication - IPC) or, similarly, Inter-Component Communication - ICC). Additionally, threats arise due to partial access control to sensitive system resources and actions from malicious applications.

This thesis, as depicted in Table 2.1 and Table 2.2, is focusing on how threats and attacks against the Android platform can be mitigated. More specifically automated tools for a) evaluating the platform's security mechanisms and b) exploiting vulnerabilities have been developed, as well as mechanisms for preventing attacks have been proposed.

Chapter 3 presents the Android operating system's architecture and all the technical features that govern it, along with all the fundamental security mechanisms that protect it at both system and application level.

Chapter 4 focuses on one of the most important security and privacy mechanism of the Android platform, which is the application permission model. According to this model, an Android application must explicitly define permissions when using sensitive API (Application Programming Interface) framework methods. The user should accept it for an application to gain access to sensitive system resources (such as geographical location or access to users' messages) or perform actions such as making phone calls or sending SMS. Understandably, there is a strong relationship between the Android Framework methods and the permissions that protect them. However, this correlation is not given as there are multiple control points of permissions throughout the Android Framework. Although Google systematically reports in the Android API documentation the permissions a method needs, deficiencies have been observed. The documentation descriptions confuse developers several times, leading to misconfigurations or misuse of permissions, which is a significant security risk for applications. Taking into account the above context and assumptions, a methodology has been developed to strictly correlate Android permissions with the methods they protect by extending the existing literature that mainly suffers from false positives due to static analysis methods limitations or operating system modifications. This methodology does not require the modification of the operating system and does not suffer from false positives, as it utilizes techniques provided by the Android Framework to create automated applications, execute them and examine if security exceptions were caused during their execution [65]. With this approach, the required permission for the proper invocation of methods can be extracted from the exception traces. The result of this work was the *Dypermin*[1] tool.

Chapter 5 investigates and redefines the Task and Activity Hijacking attacks [66]. A mechanism named *Anactijax*[2,3] has been developed to identify possible combinations of Activity attributes and Intent Flags that a malicious application can use to exploit design weaknesses of the Android Framework or applications in order to deceive users or intercept their data. To prevent this attack, a system-level mechanism was developed, named *TaskAuth*[4] (Task Authorization). This mechanism checks the ownership of the applications involved, based on the signatures and certificates they carry from their developers. For *TaskAuth*, the distinction between the Activity Hijacking attack and the Activity Injection technique is related to the ownership of the applications involved. More specifically, if an application tries to inject an Activity into a task or process of another application, it is checked whether the two applications belong to the same developer. If the verification fails, Activity hijacking is dynamically prevented while using this technique it is allowed only for system applications or between applications belonging to the same developer. The above-mentioned mechanism extends the capabilities of other methods proposed in the literature and at the same time it addresses some of their practical limitations, as it neither requires an unsafe device configuration to operate nor it involves users or developers creating complex policies that allow or prevent Activity Injection between applications. Finally, *TaskAuth* is completely transparent to the users as it does not prevent applications that use Activity Injection techniques from running without meeting the specifications it has set.

Chapter 6, focuses on the threats of Inter-Process Communication of the Android operating system and, more specifically, on Intent Redirection and Intent Hijacking [68]. Intents are the means of communication between processes on the Android platform. A component

[1]*Dypermin* Implementation: `https://gitlab.ds.unipi.gr/systems-security-laboratory/dypermin`
[2]*Anactijax* Implementation: `https://gitlab.ds.unipi.gr/systems-security-laboratory/anactijax`
[3]*Anactijax* Results: `https://gitlab.ds.unipi.gr/systems-security-laboratory/anactijax-results`
[4]*TaskAuth* Implementation: `https://gitlab.ds.unipi.gr/systems-security-laboratory/taskauth`

to communicate and exchange data with another component that belongs to the same or another application uses Intents. The Intent Redirection threat occurs when a malicious application sends data through intents to a benign application in order for the latter to process it and perform an action on behalf of the malware based on the received data. In this threat model, the benign application performs a sensitive and not a malicious action. However, receiving and using data from an untrusted application may turn the sensitive action into a malicious one. Another threat against Intents are the Hijacking attacks, where a malicious application intercepts the intents intended for a benign application. In order to prevent the above threats, a system-level security mechanism has been developed offering to users the ability to setup dynamic policies specifying the applications that are allowed to exchange data and communicate with each other. The developed mechanism is called *IntentAuth*[5] (Intent Authentication) and in addition to the creation of dynamic policies it supports authentication of transmitted data among processes. Hence, based on the user-defined trust model, *IntentAuth*, can impose end-to-end (application to application) cryptography and signature verification in order to protect the data transmitted between processes in the service-oriented Framework of Android. To achieve that, *IntentAuth* takes advantage of existing operating system mechanisms, such as SELinux and cryptographic keys, generated by Trusted Execution Environment (TEE) or Hardware Security Module (HSM). This enables *IntentAuth* to ensure the authenticity, integrity, and confidentiality of both transmitted data and the cryptographic keys of the interacting applications.

Finally, Chapter 7 concludes the research carried out and provides remarks for future work and improvements regarding the enhancement of the Android Inter-Process Communication scheme.

[5] *IntentAuth* Implementation: `https://gitlab.ds.unipi.gr/systems-security-laboratory/intentauth`

Solution	Type			Category	Enforcement Layer	Availability
	Prevention	Detection Recognition	Exploitation			
Dypermin [65]	No	Yes	No	Permission Map	Application	Yes
Anactijax [66]	No	Yes	Yes	Task Hijacking	Application	Yes
				Activity Hijacking		
TaskAuth [66]	Yes	No	No	Task Hijacking	Operating System	Yes
				Activity Hijacking		
IntentAuth [68]	Yes	No	No	Intent Hijacking	Operating System	Yes
				Malicious Activity Launch		
				Intent Redirection		

Table 2.1 Technical Contribution of the Research Work Included in this Thesis.

Research	Authors	Topic
Dypermin: Dynamic Permission Mining Framework for Android Platform, *Computers & Security* [65]	Christos Lyvas, Costas Lambrinoudakis, and Dimitris Geneiatakis	Android Operating System Security and Privacy
On Android's Activity Hijacking Prevention, *Computers & Security* [66]	Christos Lyvas, Costas Lambrinoudakis, and Dimitris Geneiatakis	Android Operating System Security and Privacy
IntentAuth: Securing Android's Intent Based Inter-Process Communication, *International Journal of Information Security* [68]	Christos Lyvas, Costas Lambrinoudakis, and Dimitris Geneiatakis	Android Operating System Security and Privacy

Table 2.2 List of Publications Directly Related to the Work of this Thesis

Even though this thesis focuses on the security of the Android operating system, in the course of the work research was also carried out in other related topics such as the security of the iOS Operating System, intrusion detection for virtual operating system environments, automated malware generation, and security on IoT environments, especially for Intelligent Transportation Systems (ITS) as Table 2.3 summarizes. The knowledge gained from the aforementioned related research activities provided useful ideas and techniques for this thesis.

Research	**Authors**	**Topic**
The Far Side of Mobile Application Integrated. Development Environments [64]	Christos Lyvas, Nikolaos Pitropakis, and Costas Lambrinoudakis	iOS Application Security
[m]allotROPism: A Metamorphic Engine for Malicious Software. Variation Development [67]	Christos Lyvas, Christoforos Ntantogian, and Christos Xenakis	Malicious Software Development
The Greater The Power, The More Dangerous The Abuse: Facing Malicious. Insiders in The Cloud [78]	Nikolaos Pitropakis, Christos Lyvas, and Costas Lambrinoudakis	Intrusion Detection System for Virtual Operating System Environments
Towards the Design of an Assurance Framework for Increasing Security and Privacy in Connected Vehicles [51]	Christos Kalloniatis, Vasiliki Diamantopoulou, Konstantinos Kotis, Christos Lyvas, KonstantinosMaliatsos, Matthieu Gay, Athanasios Kanatas, and Costas Lambrinoudakis	Intelligent Transportation Systems Security
Standardizing Security Evaluation Criteria for Connected Vehicles: A Modular Protection Profile [69]	Konstantinos Maliatsos, Christos Lyvas, Panagiotis Pantazopoulos, Costas Lambrinoudakis, Athanasios Kanatas, Matthieu Gay, and Angelos Amditis	Intelligent Transportation Systems Security
Aligning the Concepts of Risk, Security and Privacy Towards the Design of Secure Intelligent Transport Systems [35]	Vasiliki Diamantopoulou, Christos Kalloniatis, Christos Lyvas, Konstantinos Maliatsos, Matthieu Gay, Athanasios Kanatas, and Costas Lambrinoudakis	Intelligent Transportation Systems Security

Table 2.3 Contribution on Other Related Fields.

Chapter 3

Android Overview

3.1 Android Framework Internals

Android is an open-source mobile operating system (OS) officially released in 2007 by Google and the Open Handset Alliance (OHA). Since then, Android has managed to be established as the dominant operating system for mobile and other Internet of Things (IoT) devices. This is mainly because of its open-source nature that enables Original Equipment Manufacturers (OEMs) to use Android as the core operating system of their devices.

3.1.1 Android Platform Architecture

The Android platform consists of several layers, as Figure 3.1 depicts. The operating system's baseline is the reformed Linux kernel [53] responsible for orchestrating and managing hardware via drivers, memory, processes, and the Inter-Process communication mechanism based on Binder. Apart from the Android kernel, in the baseline of its architecture, resides the Secure Element, which can be the Trusted Execution Environment (TEE) that is implemented upon the System on Chip (SoC) and Central Processing Unit (CPU) system-wide approach of ARM TrustZone [46] or a dedicated Hardware Security Module (HSM) that is responsible for several critical cryptographic operations. The implementation of the Trusted Execution Environment (TEE) for Android is provided by an isolated operating system called Trusty OS [22], which is secluded from the rest of the system by hardware and software. On top of the layers mentioned above resides the Hardware Abstraction Layer (HAL) that exposes standard programming interfaces for device hardware management to the higher-level layers. On top of HAL resides the Android Runtime Environment (ART) and all the supporting Native C/C++ Libraries. In the Android platform's initial releases, during the application's execution, the JIT (Just-in-Time) compiler was responsible for translating the Dalvik Executable (DEX) application format to machine code and executed it into multiple virtual machines (Dalvik Virtual Machine - DVM) similar to the Java Virtual Machine (JVM). Where the Dalvik Virtual Machine kernel (Zygote DVM) was responsible for creating separate, secure processes in shared memory, DVM was directly deployed as the execution environment for each application. Android Runtime Environment (ART) replaced the Dalvik Virtual Machine with the Ahead-Of-Time (AOT) compilation, where the transformation occurs at the installation phase of each application. However, the use of a virtual machine as an execution environment for each application still holds. Above the Android Runtime and native libraries reside the Android System Services that provide functionality exposed by application framework APIs (Application Programming Interfaces) that communicate with system services to access the underlying hardware. Application developers may interact mostly with two groups of services: a) the system (services such as window manager and package manager) and b) media (services involved in playing and recording media). These are the services that provide application interfaces as part of the Android framework. Besides these services, there are also native services similar to Linux daemons supporting these system services, such as netd, logcatd, *etc*. Another layer of the Android architecture is the Android Application Programming Interface (API), namely Android Framework, which offers all the functionalities and features of the Android operating system to the developers for building their applications. Finally, in the top layer of the architecture reside the Android applications divided into the pre-installed system applications that have been developed by vendors or OEMs for handling critical capabilities of the device (such as SMS, phone calls, *etc.*), and the user-installed applications that can be found and installed through several distribution channels such as the official Android application market called Google Play[1] or other secondary sources such as Amazon Appstore[2] or Samsung Galaxy Store[3].

3.1.2 Android Application Components

Android applications can be developed in Java, C++, and Kotlin programming languages [1]. An AndroidPacKage (APK) archive is generated after the successful compilation of the source files that can be installed in the targeted Android device. The Java or Kotlin programming language are the recommended languages for developing Android applications over the standard Java Android SDK (Software Development Kit). At the same time, applications may contain native code as libraries with the aid of the Android Native Development Kit (NDK). Java Native Interface (JNI) is the framework that enables Java to call and be called by native appli-

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

[2] https://www.amazon.com/mobile-apps/b?ie=UTF8&node=2350149011

[3] https://www.samsung.com/global/galaxy/apps/galaxy-store/

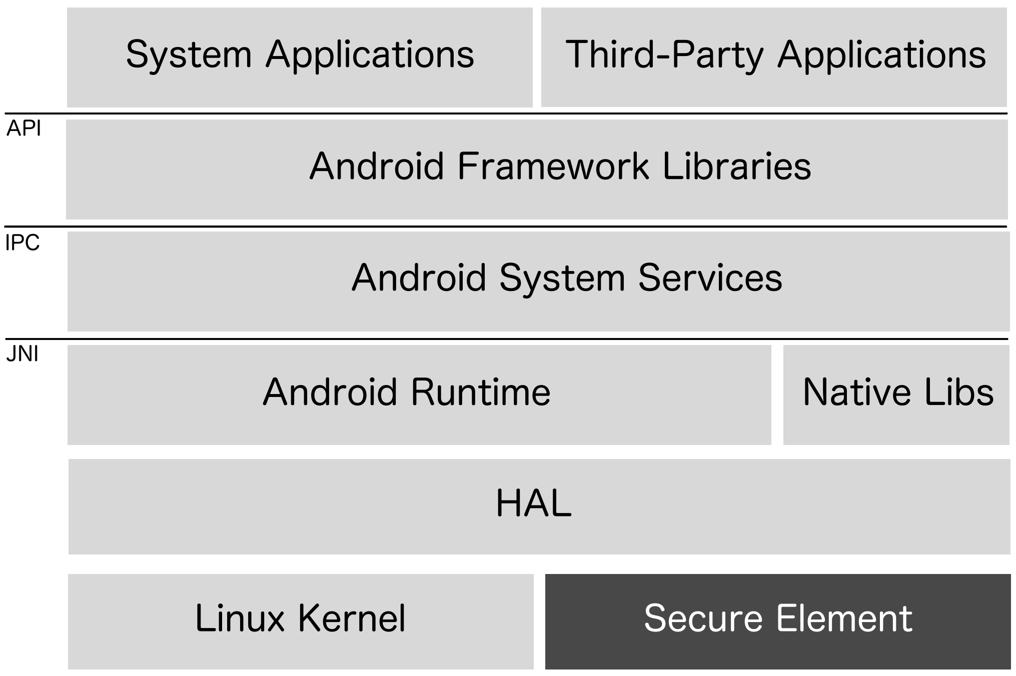


Figure 3.1 Android Operating System Architecture.

cations and libraries written in unmanaged code (C/C++). An Android application, without loss of generality, may consist of one or more combinations of the following components:

- **Activities** comprise the (UI) of Android applications and enables end user interaction with the device.
- **Broadcast Receivers** allow the system to send events to applications outside the normal execution flow, and execute in the background to handle incoming events.
- **Services** run in the background, as well, to perform a variety of computational tasks on behalf of the application.
- **Content Providers** enable applications to store internal data or share them with other Android components.

These components as well as the required permission are declared in application's `Manifest`; otherwise, the application does not function properly.

3.1.3 Android Inter-Process Communication

Android OS does not allow Android application components to exchange information directly with each other whenever it is needed. In general, Android OS relies on Inter-Process Communication (IPC) [52] or equivalently Inter–Component Communication (ICC) to enable the communication between different processes following a "server-client" model.

In fact, in Android OS, IPC is a mechanism IPC is a mechanism that enables one application to use a component of another application and synchronize their actions by using remote procedure calls (RPCs) [83, 28]. In a nutshell, Android OS has two fundamental IPC mechanisms:

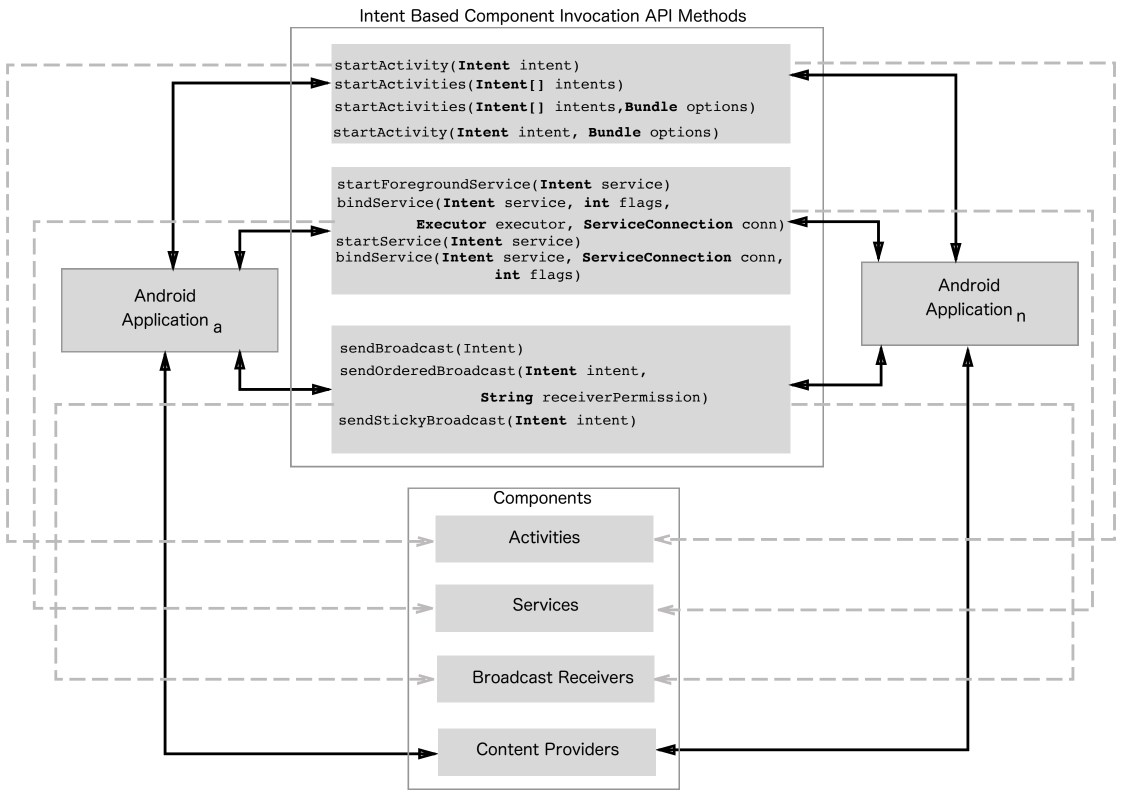
(a) the Binder [16], considered to be the core of Android's IPC mechanism, that allows app's processes to interact with other components and services at kernel level. So at low level, all communications between applications' components are established through the Binder IPC [52]. It consists of the Server, the Service Manager, the Client, and the Binder components. A client requests a service and the Server is the responsible process for providing that service. Simultaneously, the Service Manager handles the registered service [86], and

(b) the Intent [2], which provides a higher-level of abstraction that is executed over the Binder and is frequently deployed at the application layer for ICC communication [52]. Briefly, an Intent is a messaging object that carries data used to start an activity, communicates with services, or broadcasts messages [75]. In fact, intents can be used either explicitly or implicitly. In the former the application defines the target activity, service or broadcast receiver that will handle the intent, while on the latter the system or the user will determine which component should take care of it. Briefly, as Figure 3.2 depicts, an Intent is a messaging object that may carry data used to start an activity, communicate with services, or send broadcast messages [75].

Intents are divided into implicit and explicit. **Explicit Intents** (A.3) are used when the name of the class of target activity, service, or broadcast receiver that will handle the intent is defined (A.4). **Implicit Intents** (A.1) are used by the declaration of a general action to perform (A.2), rather than the class name's definition. At the same time, the system will determine the appropriate activity or broadcast receiver to handle the intent.

3.1.4 Android Task Handling

Android starts a new process [3], upon the initiation of an application's component (*i.e.*, activities), with one thread of execution. In case that another application component starts, assuming that the application has been already initiated, it is executed under the same process sharing the same thread; meaning that all components (of the same application) are executed



```
startActivity(Intent intent)
startActivities(Intent[] intents)
startActivities(Intent[] intents,Bundle options)
startActivity(Intent intent, Bundle options)

startForegroundService(Intent service)
bindService(Intent service, int flags,
        Executor executor, ServiceConnection conn)
startService(Intent service)
bindService(Intent service, ServiceConnection conn,
            int flags)

sendBroadcast(Intent)
sendOrderedBroadcast(Intent intent,
            String receiverPermission)
sendStickyBroadcast(Intent intent)
```

Figure 3.2 Android Application Core Components and Fundamental Communication Channels Among Them.

under the same process and thread, except if the application's developer has defined that specific components need to run under a separate process.

Android handles activities as a collection (of activities) named as task [4]. Usually, a task is limited within the context of an application, however, it may also contain activities from different applications [79]. Inside a task, activities are managed as a Last–in–First–Out (LIFO) [80] data structure and support only push and pop functions. So whenever an application initiates a new activity[4], the Android OS pushes the latter on the top of the stack, being in the foreground, while previous activities are pushed down in the stack without being executed. If end user presses the back button, the foreground activity is popped from the top of the stack and its execution is stopped, while the activity that is now on the top of the stack is restored. Figure 3.3 overviews activity initialization mechanism from the OS perspective.

[4]The initialization of an activity requires the invocation of the `startActivity(...)` method of the `Activity` class.

In this context, the `ActivityManagerService` (AMS)[5] class handles the life-cycle of all activities and gives to applications the capacity to communicate with it through the `ActivityManager` interface[6]. Thus for activities task management purposes the method `startActivityAsUser(...)` is called to initialise a task which finally triggers the `startActivityMayWait(...)` method, belonging to `ActivityManagerService`, and `ActivityStarter`[7] classes correspondingly.

At this point, it should be noted that the `startActivityMayWait(...)` method is fundamental for the analysis as it contains all the information concerning the assignment of activities into tasks. Specifically, it manages the `aInfo` object of `ActivityInfo` class which holds all the information regarding application's activities *i.e.*, permissions required for an activity to be executed, task affinity name, *etc.*Moreover, Android OS through its API allows (developers) to change activities' default association with a task. For instance, it might be desired the back stack to be cleared. This can be achieved by defining the appropriate activity attributes either in application's manifest or in its source code. Such attributes enable the activities to (a) assign an activity to a new task, (b) clear the current stack, and (c) relaunch an activity from the stack if already exists in it, while all the rest, on top of it in the stack, are destroyed.

3.2 Android Security Model

Android platform is a mobile operating system with several security mechanisms enforced in multiple layers, as presented in Table 3.1 shows.

[5]https://android.googlesource.com/platform/frameworks/base/+/refs/tags/android-9.0.0_r54/services/core/java/com/android/server/am/ActivityManagerService.java

[6]https://android.googlesource.com/platform/frameworks/base/+/refs/tags/android-9.0.0_r54/core/java/android/app/IActivityManager.aidl

[7]https://android.googlesource.com/platform/frameworks/base/+/refs/tags/android-9.0.0_r54/services/core/java/com/android/server/am/ActivityStarter.java

Security Mechanism	**Android Version**	**Layer of Enforcement**
Application-Level Permissions	1.0	Android Framework
Application Signing	1.0	Android Framework
Discretionary Access Control (DAC)	1.0	Kernel
No eXecute (NX)	2.3	Kernel
Random Stack Canary	4.0	Kernel
Address Space Layout Randomization (ASLR)	4.0	Kernel
Mandatory Access Control (MAC)	4.3	Kernel
Keystore System	4.3	TEE
Verified Boot	4.4	Kernel
Full Disk Encryption (FDE)	5.0	TEE
		OS
Gatekeeper	6.0	TEE
		OS
File Based Encryption (FBE)	7.0	OS
		TEE or HSM
Strongbox Keymaster System	9.0	HSM
Weaver	9.0	HSM
		OS

Table 3.1 Primary Android Security Mechanisms and Their Enforcement Point.

3.2.1 System Security

Each application (as a UNIX-based process) is executed in an isolated sandbox environment, separated from other processes. Android OS assigns a different User Identifier (UID)[8] and a directory to each application (except for applications coming from the same developer where a UID can be shared). At the same time, all its files are protected against unauthorized access from other applications in such a way that only the assigned UID can access them. More precisely, the UNIX–based permission model is enforced by the Kernel through the Uniform Resource Identifier (URI) [71] permissions that allow an application to application interaction, giving the ability to an application to grant selective access to data resources it owns.

In addition to the aforementioned Discretionary Access Control (DAC) mechanism, in the Android platform a Mandatory Access Control (MAC) has been implemented upon SELinux since Android 4.3. The MAC is enforced by system-level security policies that enforce the default deny [5]. Thus, SELinux provides a centralized policy configuration approach stronger than DAC for isolating and sandboxing Android applications and privileged Android system daemons. More precisely, it uses roles based on identities to limit their access to domains (processes) or types (for objects, *i.e.*, , files, folders, *etc.*). Moreover, SELinux grants permissions based on UIDs or other security-relevant information [71] about each process where only explicitly defined actions can be performed.

Keymaster implements the Android key store into the Trusted Execution Environment (TEE). This mechanism stores cryptographic keys into TEE and protects them from unauthorized access, and even if a process is compromised (including Kernel), an attacker cannot read key material stored in Keymaster. Keystore [19] is the crypto–service API in the Android framework based on Keymaster Hardware Abstraction Layer (HAL). The security of Android Keystore is highly dependent on SELinux enforcement, and dedicated policies[9] safeguard it. In the Android framework, a process can gain access to other processes Keystore if this is explicitly defined. This is a standard method whenever a privileged process is dependent upon or manages other processes. Such an example is the defined rules that allow system services to handle the keys of VPN (Virtual Private Network), Bluetooth, and WiFi services based on their UIDs. Another implementation of the Android key store is the Strongbox imported into Android version 9.0 that implements the Android Keymaster Hardware Abstraction Layer

[8]https://android.googlesource.com/platform/system/core/+/refs/tags/android-11.0.0_r3/libcutils/include/private/android_filesystem_config.h

[9]https://android.googlesource.com/platform/system/security/+/refs/tags/android-11.0.0_r3/keystore/permissions.cpp

(HAL) into a dedicated tamper-resistant hardware security module (HSM) for supported devices.

The verified boot is the security mechanism that provides a chain of trust since the introduction of Android 4.4 by ensuring the firmware's integrity and authenticity from the `bootloader` to the `boot`, `system`, `vendor` and `OEM` partitions with the aid of TEE enforced at the kernel level [71]. More specifically, during the device boot, every stage verifies the upcoming stage's integrity before its execution.

The Gatekeeper [17] or Weaver implements the authentication of the end-user ownership of the device. The first case implements user lock screen verification based on PIN or password or pattern with Keymaster for TEE, while the latter with HSM upon Strongbox [71]. After the successful end-user authentication, Gatekeeper or Weaver communicates this to Keymaster or Strongbox to grant access to authentication-related keys.

Data encryption at rest was implemented initially in Android 4.4 with the Full Disk Encryption (FDE) mechanism that uses a protected credential key to encrypt the entire user data partition. The use of Trusted Execution Environment's (TEE) signing capability was firstly introduced in Android 5.0. This mechanism's main disadvantage was the core device services' inability to operate until the device's end-user provided a valid password entry [71]. The Full Disk Encryption mechanism was replaced by the File-Based Encryption (FBE) mechanism in Android 7.0 that leverages TEE or HSM to protect individual files with credentials by different users. File-Based Encryption provides cryptographical protection of per-user data on Android devices.

3.2.2 Application Security

Another class of permissions in the Android platform is the Application-Level Permissions or more specific Android permissions [20]. In this model, applications should "request" the suitable permissions to interact with sensitive system services, other applications, and components. This means that programmers should declare the necessary permissions (both normal and dangerous ones) in the application's manifest file if they target devices relying on Android Lollipop and previous versions in order to gain access to protected resources. From Android 6.0 onwards, programmers should define the permissions at the application's manifest file and in cases where "dangerous" permissions [20] are used to protect a resource these permissions should be requested at run time. The permissions requested by applications are granted by users either during the installation of the application or at runtime, depending on the Android version that they rely on. For instance, if an application reads the device ID (*e.g.*, IMEI), it requires the `READ_PHONE_STATE` permission to be defined in its manifest file and the user to grant it at runtime otherwise, a security exception will be raised during execution.

Android application signing [21] is a mechanism enforced in the Android Framework to ensure that only applications from legitimate sources can be executed into a mobile device. Applications from non–trusted sources can be executed into Android devices only when the developer option is enabled for testing purposes. As part of the System Service, the Package Manager checks the application's signature's validity based on their carrying certificate during installation. Currently, Android does not provide Certificate Authority (CA) verification for application certificates, and all of them are self–signed and generated by OEMs, vendors, or developers. With this Public Key Infrastructure (PKI) oriented mechanism, Android OS can ensure each application's ownership onto a device. After the successful signature validation process, the system assigns UIDs into applications and enforces the isolation sandbox. An application that shares the same certificate with others can share the same UID if this is explicitly defined in its manifest file. In addition, applications that share the same UID can share resources such as preferences, TEE or Strongbox keys, etc.

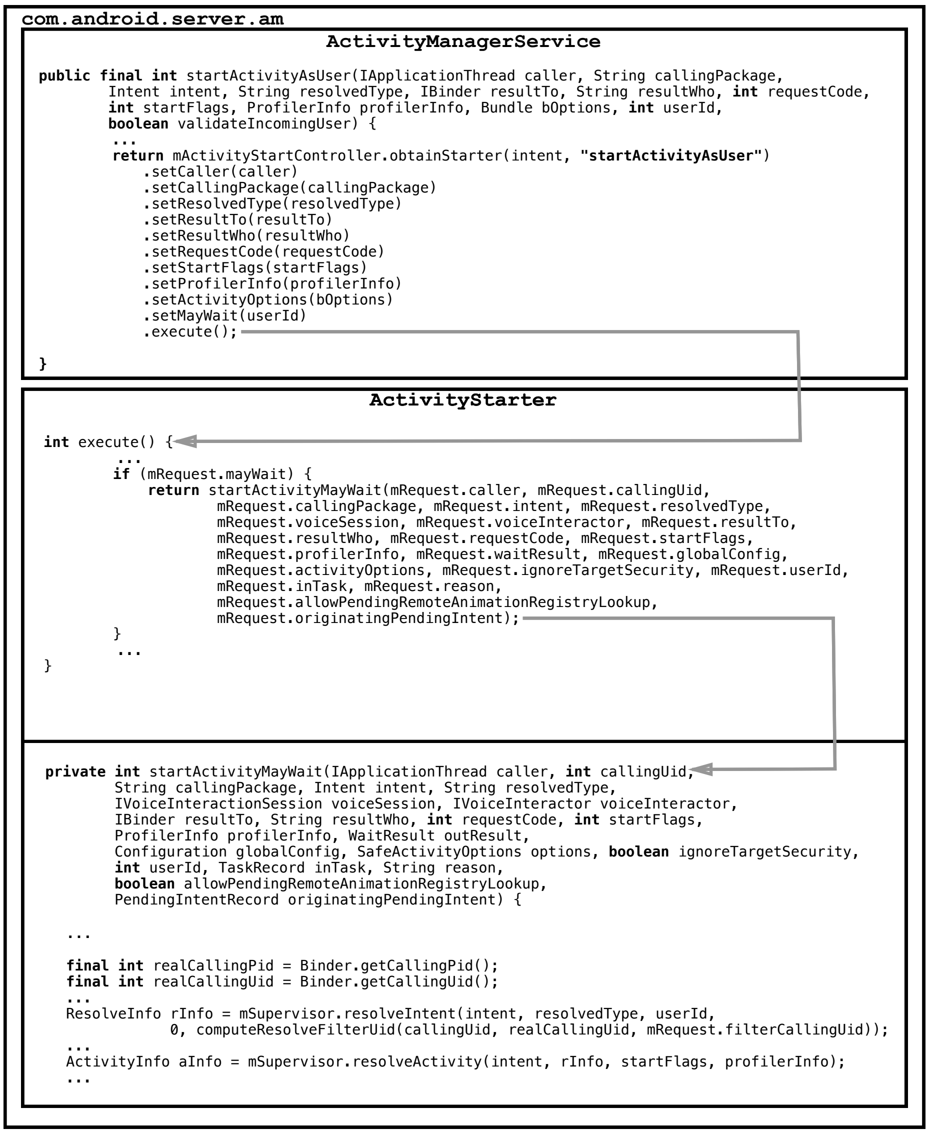
3.2.3 Exploit Prevention

The most effective security mechanisms regarding the prevention of stack and heap-based vulnerabilities exploitation on Android are presented here, while more comprehensive lists can be found in [27, 6, 8–15, 7].

Stack Canary is a prevention mechanism that usually places random bytes (`QWORD`) based on `/dev/urandom` into a stack frame before the return address of an invoked function and checks if the contents are changed before the function returns. The android platform uses random canary since the introduction of version 4.0 [36].

Hardware-based No eXecute (NX) prevents code execution on the stack and heap, and it was introduced in Android version 2.3 [6]. With this mechanism, memory sections (pages) of the process which contain code are marked as executable and read-only [45]. In other words, a memory page can be marked as executable or writable.

Another prevention mechanism against code–reusing attacks that complement the NX mechanism was imported into the Android platform in version 4.0 called Address Space Layout Randomization (ASLR). This mechanism randomizes the locations of memory areas within an address space to make it probabilistically difficult for an attacker to gain control of a process [60].



```
com.android.server.am
                    ActivityManagerService

public final int startActivityAsUser(IApplicationThread caller, String callingPackage,
        Intent intent, String resolvedType, IBinder resultTo, String resultWho, int requestCode,
        int startFlags, ProfilerInfo profilerInfo, Bundle bOptions, int userId,
        boolean validateIncomingUser) {
        ...
        return mActivityStartController.obtainStarter(intent, "startActivityAsUser")
            .setCaller(caller)
            .setCallingPackage(callingPackage)
            .setResolvedType(resolvedType)
            .setResultTo(resultTo)
            .setResultWho(resultWho)
            .setRequestCode(requestCode)
            .setStartFlags(startFlags)
            .setProfilerInfo(profilerInfo)
            .setActivityOptions(bOptions)
            .setMayWait(userId)
            .execute();

}
```

```
                    ActivityStarter

int execute() {
        ...
        if (mRequest.mayWait) {
            return startActivityMayWait(mRequest.caller, mRequest.callingUid,
                    mRequest.callingPackage, mRequest.intent, mRequest.resolvedType,
                    mRequest.voiceSession, mRequest.voiceInteractor, mRequest.resultTo,
                    mRequest.resultWho, mRequest.requestCode, mRequest.startFlags,
                    mRequest.profilerInfo, mRequest.waitResult, mRequest.globalConfig,
                    mRequest.activityOptions, mRequest.ignoreTargetSecurity, mRequest.userId,
                    mRequest.inTask, mRequest.reason,
                    mRequest.allowPendingRemoteAnimationRegistryLookup,
                    mRequest.originatingPendingIntent);
        }
        ...
}
```

```
private int startActivityMayWait(IApplicationThread caller, int callingUid,
        String callingPackage, Intent intent, String resolvedType,
        IVoiceInteractionSession voiceSession, IVoiceInteractor voiceInteractor,
        IBinder resultTo, String resultWho, int requestCode, int startFlags,
        ProfilerInfo profilerInfo, WaitResult outResult,
        Configuration globalConfig, SafeActivityOptions options, boolean ignoreTargetSecurity,
        int userId, TaskRecord inTask, String reason,
        boolean allowPendingRemoteAnimationRegistryLookup,
        PendingIntentRecord originatingPendingIntent) {

   ...

   final int realCallingPid = Binder.getCallingPid();
   final int realCallingUid = Binder.getCallingUid();
   ...
   ResolveInfo rInfo = mSupervisor.resolveIntent(intent, resolvedType, userId,
             0, computeResolveFilterUid(callingUid, realCallingUid, mRequest.filterCallingUid));
   ...
   ActivityInfo aInfo = mSupervisor.resolveActivity(intent, rInfo, startFlags, profilerInfo);
   ...
```

Figure 3.3 Android (Version 9) Activity Initialization and Task Handling.

Chapter 4

Android Permission Model

Android Permission Model is a vital security mechanism for end user's privacy protection. Programmers get information on the correlation between permissions and protected Application Programming Interface (API) methods through Android's documentation. However a full correlation between the Android's Software Development Kit (SDK) API methods and permissions doesn't exist. This is caused by the fact that (a) documentation may accidentally lack information, and (b) Android has hidden and internal API methods that are not directly accessible at the application layer since they are not included in the Development SDK. Though the latter cannot be directly accessed there are several publicly available sources that give guidelines[1] on how to gain access to such resources [58]. So eventually, programmers can gain access to these hidden API methods.

Moreover, as the Android OS evolves and in order to improve end users' experiences, it proceeds with various modifications to the underlying subsystems. For instance, permissions are enabled dynamically on the latest versions of Android, while from version 6.0 backwards they were granted statically. In addition, some API methods are deprecated, while other are introduced to support additional functionalities. So it is evident that these types of changes not only affect the API methods and permission mapping but also introduce inconsistencies in it among different Android versions according to [100].

4.1 System Level Permission Enforcement

Android utilizes the Application-Level Permission model to restrict access to device's sensitive features from untrusted applications. Specifically, Android as a service oriented framework enforces permissions' control at the lower stack layers; at Android framework level [82], without

[1]Android Hidden APIs: https://github.com/anggrayudi/android-hidden-api

providing a centralized point for accomplishing this task. For instance, at the Android framework layer there are several API methods (`enforceCallingOrSelfPermission(...)`, `enforcePermission(...)`, `enforceCallingPermission(...)`, *etc.*,) that impose permission controls declared in the `Context`[2] Interface which is responsible to expose methods for accessing the underlying resources.

Figure 4.1 illustrates a high level example of a permission enforcement procedure. According to this example the application invokes the `getIsimImpi()`[3] API method using Java reflection technique (this specific method cannot be found in the public documentation of the Android framework as it is not part of the public API) which is responsible to call the relevant resource implemented in the service side. In this specific case, the latter performs a permission control through the `enforceCallingOrSelfPermission()` API method. If the application has not defined the corresponding permissions (*i.e.*, `READ_PRIVILEGED_PHONE_-STATE`) the framework will cause a security exception reporting in the exception stack the missing permission. Note that not all the enforcement points raise such a security exception if a permission is missing; for instance this is the case of `checkPermission()` Framework method.

4.2 Android Application Permission Acquisition

Access to any sensitive resource of Android Framework is granted through a protected Application Programming Interface (API) method. An application in order to use a protected API method it must first declare the corresponding permissions in its manifest file, and request it also at runtime if it is executed on Android latest versions, otherwise a security exception is raised. In any case, users should give their consent for the permissions requested by the application either during the first time that a protected API method is invoked or during installation process, depending on the Android version [38, 100].

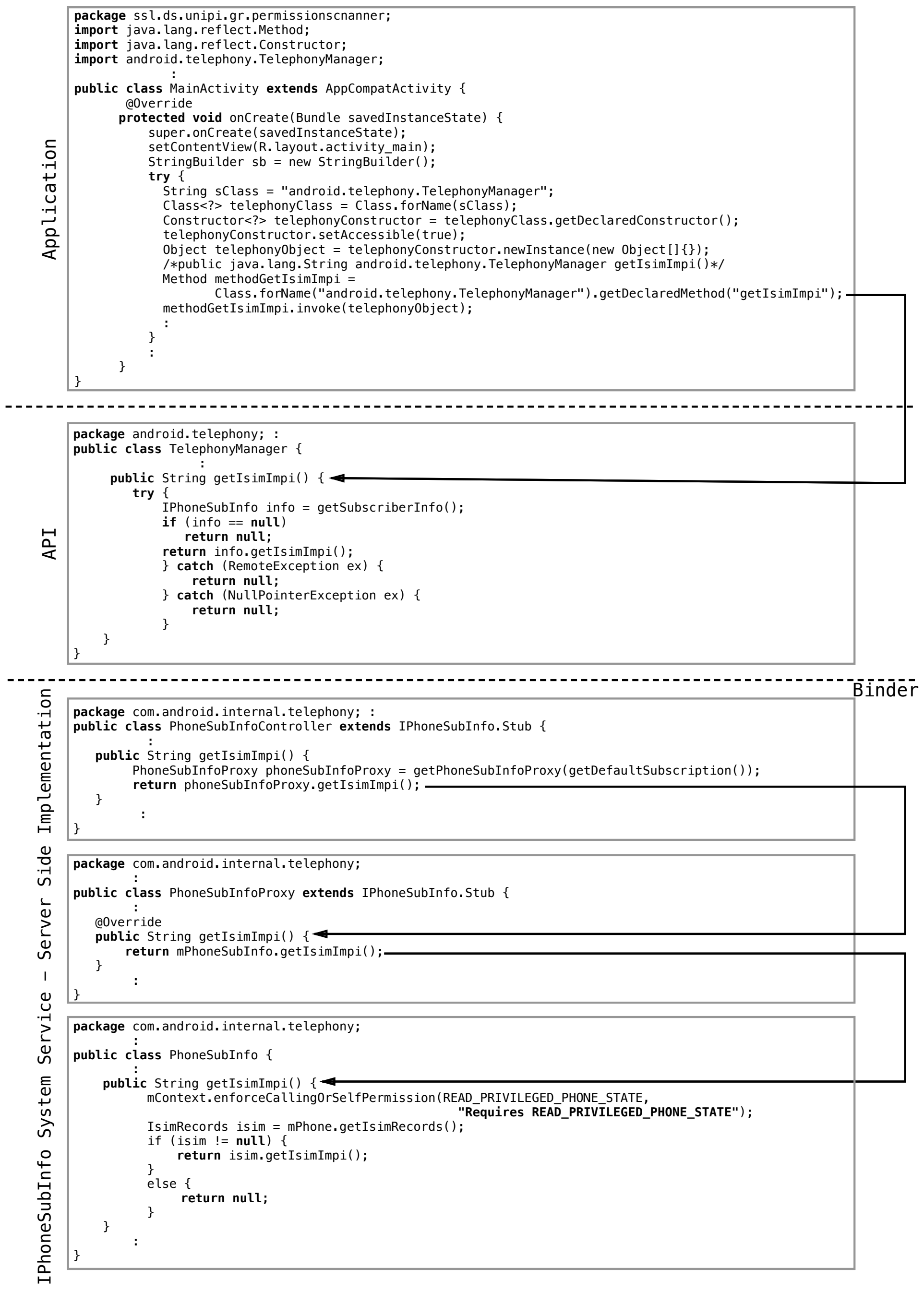
4.3 Threats of Android Permission Model

4.3.1 Permission Re-Delegation

The threat of permission re-delegation is a form of confused deputy attack [47, 91] or similarly privilege escalation attack. In the Android platform, the threat of permission re-delegation

[2] `https://android.googlesource.com/platform/frameworks/base.git/+/refs/tags/android-7.0.0_r34/core/java/android/content/Context.java`

[3] `https://android.googlesource.com/platform/frameworks/base.git/+/refs/tags/android-7.0.0_r34/telephony/java/android/telephony/TelephonyManager.java`



```
package ssl.ds.unipi.gr.permissionscnanner;
import java.lang.reflect.Method;
import java.lang.reflect.Constructor;
import android.telephony.TelephonyManager;
             :
public class MainActivity extends AppCompatActivity {
       @Override
      protected void onCreate(Bundle savedInstanceState) {
          super.onCreate(savedInstanceState);
          setContentView(R.layout.activity_main);
          StringBuilder sb = new StringBuilder();
          try {
            String sClass = "android.telephony.TelephonyManager";
            Class<?> telephonyClass = Class.forName(sClass);
            Constructor<?> telephonyConstructor = telephonyClass.getDeclaredConstructor();
            telephonyConstructor.setAccessible(true);
            Object telephonyObject = telephonyConstructor.newInstance(new Object[]{});
            /*public java.lang.String android.telephony.TelephonyManager getIsimImpi()*/
            Method methodGetIsimImpi =
                   Class.forName("android.telephony.TelephonyManager").getDeclaredMethod("getIsimImpi");
            methodGetIsimImpi.invoke(telephonyObject);
            :
          }
          :
      }
}
```

```
package android.telephony; :
public class TelephonyManager {
                 :
     public String getIsimImpi() {
        try {
            IPhoneSubInfo info = getSubscriberInfo();
            if (info == null)
               return null;
            return info.getIsimImpi();
            } catch (RemoteException ex) {
               return null;
            } catch (NullPointerException ex) {
               return null;
            }
    }
}
```

```
package com.android.internal.telephony; :
public class PhoneSubInfoController extends IPhoneSubInfo.Stub {
          :
   public String getIsimImpi() {
        PhoneSubInfoProxy phoneSubInfoProxy = getPhoneSubInfoProxy(getDefaultSubscription());
        return phoneSubInfoProxy.getIsimImpi();
   }
         :
}
```

```
package com.android.internal.telephony;
        :
public class PhoneSubInfoProxy extends IPhoneSubInfo.Stub {
        :
   @Override
   public String getIsimImpi() {
       return mPhoneSubInfo.getIsimImpi();
   }
        :
}
```

```
package com.android.internal.telephony;
        :
public class PhoneSubInfo {
        :
    public String getIsimImpi() {
          mContext.enforceCallingOrSelfPermission(READ_PRIVILEGED_PHONE_STATE,
                                                  "Requires READ_PRIVILEGED_PHONE_STATE");
          IsimRecords isim = mPhone.getIsimRecords();
          if (isim != null) {
              return isim.getIsimImpi();
          }
          else {
               return null;
          }
    }
        :
}
```

Figure 4.1 Android (Version 7) Permission Enforcement Control Point Example.

occurs whenever an application with granted permission performs an action on behalf of a less privileged application [42]. This kind of threat can be materialised in several ways: (a) An application may intentionally expose sensitive functionality, and a malicious application leverages it in order to perform a non-legitimate action. (b) A developer may accidentally expose a sensitive functionality that can be exploited by an intrusive application. In any of the cases mentioned above, permission re-delegation is a threat with a severe impact on end users' privacy because the Android framework does not restrict application with fewer permissions to access more privileged application's components [32]. This type of attacks is primarily linked with design errors [49] by the application developers that ignore security recommendations and best practices suggestions.

4.3.2 Over-Privileged Android Applications

An application is considered over-privileged whenever it declares more than the needed permissions for its normal functionality [41, 44]. This could increase vulnerabilities as well as security and privacy risks in those applications [90]. The main causes of over-privileged applications are development misconfigurations, poor documentation regarding the correlation between Android permissions and API methods, or lack of knowledge on application developers' behalf about application security development best practices. Several research works have tried to detect over-privileged applications with static [24, 26], dynamic [41], or hybrid analysis [44].

4.4 Android Permission Model Mapping

The correlation between Android permissions and API methods is defined as Permission Map and is undoubtedly a vital component for Android's application security assessment *e.g.*, overprivilleges [24, 26, 94, 41, 44], malware detection [23, 77, 57], *etc.*, since it can identify inconsistencies, misconfigurations and other flaws. Thus, several researchers embrace either a static or a dynamic approach in order to correlate Android APIs with the corresponding permissions, and construct the so-called permission map.

Solutions adopting the static approach perform a full analysis [24, 26, 25] of the Android OS source code. Though such solutions can achieve high coverage, they don't have a way to validate their outcomes; in other words, they suffer of false positives pairs. On top of that, in many cases [24, 26] they require the identification of Android's permission enforcement functions beforehand, otherwise their analysis fails. Or other's scope of analysis is quite limited despite their accuracy [29]. Approaches adopting the dynamic analysis [41, 65] are free of false positives but, depending on their characteristics, may not achieve to cover the full set of protected API methods. The latter it is also limitation of [65] which in fact cannot successfully generate permission mapping for APIs methods requiring specific conditions to be met i.e., in case where an API method requires a previous event to be triggered for its execution afterwards. The comparison among the proposed methods regarding the Android permission mapping generation is listed in Table 4.1.

4.4.1 Dynamic Analysis Based Approaches

Stowaway [41] is the very first work that develops a map between protected APIs and permissions in the context of detecting over-privileged applications. Stowaway builds a solution considering the advantages of Java tests and Randoop [76] in order to compile the appropriate tests and dynamically invoke the APIs incorporated in a given application. However, Stowaway needs modifications to the underlying OS in order to monitor the permission enforcement mechanism and thus identify which permission is required for a specific Android's API. This method achieves 85% coverage of the Android API, but it also requires manual testing for the permission map identification and manual effort to be adaptable to a variety of Android Open Source Project (AOSP) versions. It is important to stress that Stowaway is no longer supported and thus the provided permission map is totally outdated[4].

Lyvas *et al.* [65] introduced an open-source orthogonal to the existing approaches framework, named *Dypermin*, which builds on the advantages of the Android framework features to generate the permission map. *Dypermin* relies on: (a) the security exceptions raised by the Android OS during the execution of an application that lacks the necessary permissions for invoking protected APIs, (b) the capability to access any protected API *i.e.*, publicly accessible or hidden, through the Android framework, and (c) the Java reflection technique for invoking protected APIs at run time.

Using the aforementioned features, *Dypermin* generates and invokes the appropriate objects and APIs embedded in an application, and inspects whether a security exception is raised. In this way, no modifications to the underlying Android OS is required and thus the proposed methodology can be applied to any unmodified Android device, emulator or Virtual Machine without needing root access.

As depicted in Figure 4.2 *Dypermin*'s operation can be abstracted in three phases. Specifically, during the first phase (steps 1-2) *Dypermin* extracts from the Android framework all the packages and their classes with their corresponding APIs, including additional related information such as arguments, modifiers, class members, returning values as well as annotations and comments, and stores them in a collection named SDK dump. Moreover, during this phase *Dypermin* processes also API's documentation; that is the Javadoc comments of API's methods, and identifies if a given method is publicly accessible or hidden (based on "@hide" document annotations). Additionally, it extracts all the permissions per method that

[4]http://www.android-permissions.org

are defined in comments or method annotations. In this way, *Dypermin* is able to compile the complete list of available methods.

To generate a signature for any available method of the Android framework, *Dypermin* utilizes a custom made tool based on the javalang python parser package.

In this point it is crucial to recall that the Android framework contains both publicly accessible and hidden protected API methods, and also that the hidden API methods are excluded from the Android development SDK library (`android.jar`), meaning that they cannot be directly invoked by an application. To bypass this restriction developers have two options:

1. either use the Java reflection mechanism that enables an application to load, at runtime, the corresponding API from the devices `framework.jar` classes, or

2. replace the original Android development SDK library (`android.jar`) with a custom one that includes all Android APIs.

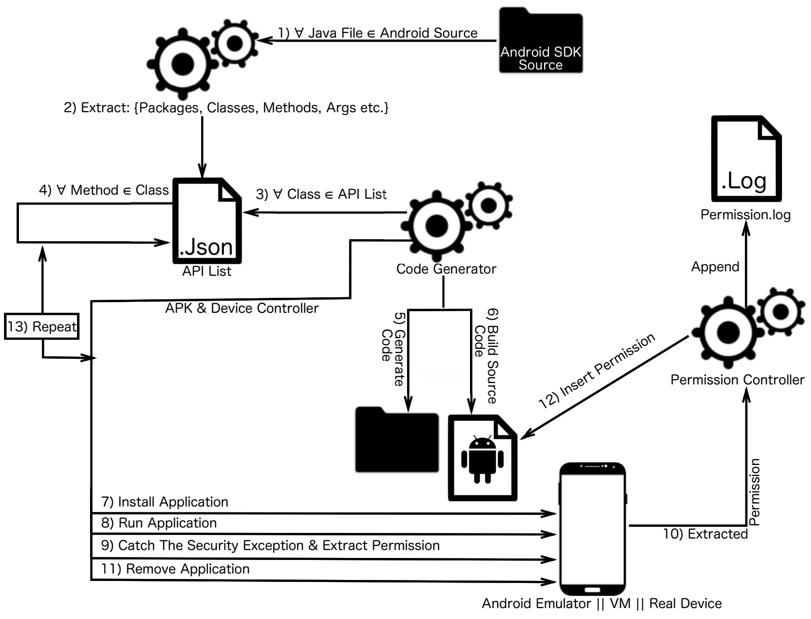


Figure 4.2 *Dypermin's* Design for Permission Map Compilation

The proposed framework, has adopted the Java reflection approach since it is transparent to the underlying OS and it does not pose a need for modifying the device's OS.

During the second phase (steps 3-5), *Dypermin* identifies all the available constructors for the corresponding APIs, using the SDK dump collection that was generated during the first phase. *Dypermin* in order to instantiate an object of a given class first prioritize the candidate constructors based on the type of arguments that they contain in the following way: (a) Firstly, constructors with arguments that belong to Java basic datatypes (int, long, char *etc.*) or belong to `android.content.Context` and/or `java.lang.String` class. (b) Secondly, constructors with basic datatypes or arguments from Context and/or from classes that belongs to java.lang package. (c) Thirdly comes constructors that they dont' require arguments. (d) Finally, any constructor that do not contain arguments from abstract classes or belong to interfaces.

Dypermin in order to instantiate the objects of a given constructor, it first generates the object instantiation code of the involved arguments in a recursive way, as Listing 4.1 depicts.

If more than one available constructor belongs in the same category *Dypermin* prioritize them in those that belongs to the publicly accessible API and then to those from the hidden API. Using such an approach it is possible to generate automatically valid code for almost any object. Furthermore it should be stressed that in cases where constructors or methods of the corresponding classes require access to a valid `Context` class object, an instance of an application context is passed. Otherwise, object instantiation either fails or it does not trigger a security exception, since the context object is null. For method invocation where objects are required as parameters, *Dypermin* generates them in the same way as above.

In cases where classes have constructors with abstract class or interface arguments the aforementioned heuristic cannot be applied for their instantiation. This is leveraged by the fact that in most cases the required constant values associated with the manager class are hardcoded in the class code. *Dypermin* is able to parse the under-analysis class code and extract those tokens and use them to create a valid object of the given class.

During the third phase, *Dypermin* employs the toolkit Gradle [18] for building applications (step 6), that *call* (through the source code generated in the second phase) the methods that are necessary *i.e.*, (`onCreate()` method - see Figure 4.1) for the invocation of the examined APIs. These applications do not specify any permissions in their manifest files and handle security exceptions in a `try ... catch` statement. For every security exception all relative data are kept in a log file. The applications are installed and executed (steps 7-8), through the Android Debug Bridge (adb), the APIs under examination are invoked, and the status is monitored through the analysis of the log files. Utilizing the fact that the Android framework raises a security exception if a protected API is invoked without the appropriate permissions appearing in the application's manifest file, *Dypermin* identifies the permission that caused the security exception and correlates it with the examined API (step 10).

In step 9-10, *Dypermin* defines the identified missing permissions in the application's manifest file and repeats the execution of the applications until no more security exceptions appear. In this way *Dypermin* can also identify protected APIs that are correlated with multiple permissions. *Dypermin* revokes the application (step 11) from the device after the first successful invocation of the examined API.

The above-mentioned method is also suitable for mining multiple permission for a given API method. The high-level overview of this functionality is as follows. For every method invocation that triggers a `Security Exception` the *Dypermin* Proof-of-concept implementation would be able to import it in the application's manifest file and repeat this process until no new security exceptions occur. This practice is able to discover all needed permissions (if multiple permissions required) for any method (as long as a security exception occurs).

An example of *Dypermin's* auto generated application code is depicted in Listing 4.1. In particular, in line 7 the method `getSystemService(..)` was used to retrieve of the system-level service by its name, with a new instance of the class `ConnectivityManager`[5], able to handle the management of network connections. From line 11 to 75 *Dypermin* generates in a recursive way, the object instantiation code of the involved arguments of the method `registerNetworkAgent(..)`. The method to be invoked is declared by its name and its parameter's signature (lines 80-86). Then all the generated class objects (lines 11-75) are used to call the method `registerNetworkAgent(..)` (lines 87-92). If any security exceptions are raised they will be handled at the catch block (lines 100 to 135), identifying the permissions that are missing from the application's manifest file.

```
1
2  protected void onCreate(Bundle savedInstanceState) {
3    super.onCreate(savedInstanceState);
4    setContentView(R.layout.activity_main);
5    StringBuilder sb = new StringBuilder();
6    try{
7        ConnectivityManager object_aaed329c =
8                (ConnectivityManager)
9                        getSystemService(Context.CONNECTIVITY_SERVICE);
10     try{
11       /*public android.os.Handler Handler(boolean)*/
12       String string_12b34e3b =  "android.os.Handler";
13       Class<?> class_12b34e3b =
14               Class.forName(string_12b34e3b);
15       Constructor<?> constructor_12b34e3b =
16               class_12b34e3b.getDeclaredConstructor(Boolean.TYPE);
17       Object object_12b34e3b = constructor_12b34e3b.newInstance(true);
18       /* public android.os.Messenger Messenger(android.os.Handler)*/
19       String string_3f76bc12 = "android.os.Messenger";
20       Class<?> class_3f76bc12 = Class.forName(string_3f76bc12);
```

[5]https://android.googlesource.com/platform/frameworks/base.git/+/refs/tags/android-7.0.0_r34/core/java/android/net/ConnectivityManager.java

```
        Constructor<?> constructor_3f76bc12 =
                class_3f76bc12.getDeclaredConstructor(Handler.class);
        Object object_3f76bc12 =
                constructor_3f76bc12.newInstance((Handler) object_12b34e3b);
        /* public java.lang.String String(int[], int, int)*/
        String string_e65b2e74 = "java.lang.String";
        Class<?> class_e65b2e74 = Class.forName(string_e65b2e74);
        Constructor<?> constructor_e65b2e74 =
                class_e65b2e74.getDeclaredConstructor(int[].class,
                        Integer.TYPE, Integer.TYPE);
        Object object_e65b2e74 =
                constructor_e65b2e74.newInstance(new int[]{1, 1, 1}, 1, 1);
        /* public java.lang.String String(int[], int, int)*/
        String string_440aa7cf = "java.lang.String";
        Class<?> class_440aa7cf =
                Class.forName(string_440aa7cf);
        Constructor<?> constructor_440aa7cf =
                class_440aa7cf.getDeclaredConstructor(int[].class,
                        Integer.TYPE, Integer.TYPE);
        Object object_440aa7cf =
                constructor_440aa7cf.newInstance(new int[]{1, 1, 1}, 1, 1);
        /* public android.net.NetworkInfo NetworkInfo(int, int,
        java.lang.String, java.lang.String)*/
        String string_895891de = "android.net.NetworkInfo";
        Class<?> class_895891de = Class.forName(string_895891de);
        Constructor<?> constructor_895891de =
                class_895891de.getDeclaredConstructor(Integer.TYPE,
                Integer.TYPE,
                String.class , String.class );
        Object object_895891de =
                constructor_895891de.newInstance(1, 1,
                (String) object_e65b2e74, (String) object_440aa7cf);
        /* public android.net.LinkProperties LinkProperties()*/
        String string_9fc32827 = "android.net.LinkProperties";
        Class<?> class_9fc32827_0591_4521_b76e_0481d35c7905 =
                Class.forName(string_9fc32827);
        Constructor<?> constructor_9fc32827 =
                class_9fc32827_0591_4521_b76e_0481d35c7905.
                        getDeclaredConstructor(null);
        Object object_9fc32827 =
                constructor_9fc32827.newInstance(new Object[]{});
        /* public android.net.NetworkCapabilities NetworkCapabilities()*/
        String string_78b4abb2 = "android.net.NetworkCapabilities";
        Class<?> class_78b4abb2 = Class.forName(string_78b4abb2);
```

```
    Constructor<?> constructor_78b4abb2 =
            class_78b4abb2.getDeclaredConstructor(null);
    Object object_78b4abb2 =
            constructor_78b4abb2.newInstance(new Object[]{});
    /* public android.net.NetworkMisc NetworkMisc()*/
    String string_6998d15e = "android.net.NetworkMisc";
    Class<?> class_6998d15e = Class.forName(string_6998d15e);
    Constructor<?> constructor_6998d15e =
            class_6998d15e.getDeclaredConstructor(null);
    Object object_6998d15e =
            constructor_6998d15e.newInstance(new Object[]{});
    /*public int android.net.ConnectivityManager
    registerNetworkAgent(android.os.Messenger,
    android.net.NetworkInfo, android.net.LinkProperties,
    android.net.NetworkCapabilities, int, android.net.NetworkMisc)*/
    Method methodregisterNetworkAgent =
            Class.forName("android.net.ConnectivityManager").
                    getDeclaredMethod("registerNetworkAgent",
                    Messenger.class, NetworkInfo.class,
                    LinkProperties.class,
                    NetworkCapabilities.class,
                    Integer.TYPE, NetworkMisc.class);
    methodregisterNetworkAgent.invoke(object_aaed329c,
            (Messenger) object_3f76bc12,
            (NetworkInfo) object_895891de,
            (LinkProperties) object_9fc32827,
            (NetworkCapabilities) object_78b4abb2,1,
            (NetworkMisc) object_6998d15e);
    System.out.println("android.net.ConnectivityManager public " +
            "int registerNetworkAgent(android.os.Messenger," +
            "android.net.NetworkInfo, android.net.LinkProperties, " +
            "android.net.NetworkCapabilities," +
            "int, android.net.NetworkMisc)" + " No Error");
  }
  catch (Exception e) {
    StringWriter sw = new StringWriter();
    PrintWriter pw = new PrintWriter(sw);
    e.printStackTrace(pw);
    String error = sw.toString();
    String pattern1 = ".permission.[A-Z_]+";
    String pattern2 = "[cC]arrier [Pp]rivilege";
    String pattern3 = "java.lang.SecurityException: Requires ([A-Za-z_]+)";
    Pattern r1 = Pattern.compile(pattern1);
    Pattern r2 = Pattern.compile(pattern2);
```

```
      Pattern r3 = Pattern.compile(pattern3);
      Matcher m1 = r1.matcher(error);
      Matcher m2 = r2.matcher(error);
      Matcher m3 = r3.matcher(error);
      if (m1.find()) {
          sb.append("android.net.ConnectivityManager public int " +
                  "registerNetworkAgent(android.os.Messenger," +
                  "android.net.NetworkInfo, android.net.LinkProperties, " +
                  "android.net.NetworkCapabilities, int," +
                  "android.net.NetworkMisc) " +  m1.group(0));
      }else if (m2.find()) {
          sb.append("android.net.ConnectivityManager public int" +
            "registerNetworkAgent(android.os.Messenger," +
            " android.net.NetworkInfo, " +
            "android.net.LinkProperties," +
            "android.net.NetworkCapabilities, int, android.net.NetworkMisc) " +
            m2.group(0));
          System.out.println("Permission-Extracted:" + " " + sb.toString());
      }else if (m3.find()) {
          sb.append("android.net.ConnectivityManager public int" +
              " registerNetworkAgent(android.os.Messenger," +
              "android.net.NetworkInfo, android.net.LinkProperties, " +
              "android.net.NetworkCapabilities, " +
              "int, android.net.NetworkMisc) " + "android.permission." +
              m3.group(1));
      }else{
          e.printStackTrace();
      }
    }
  }
  catch (Exception e) {
      e.printStackTrace();
  }
}
```

Listing 4.1 Code Generated by *Dypermin* For `registerNetworkAgent(...)` Private API Method.

4.4.2 Static Analysis Based Approaches

While only Stowaway [41] and *Dypermin* [65] utilize dynamic analysis, there are few other methods such as PScout [24], Bartel *et al.*[26], and Axplorer [25] that use static analysis of Android OS source code to derive the permission map.

PScout [24] is the first tool that could perform Java bytecode analysis with the aid of Soot framework [87] and generate a control flow graph (CFG) based on Class Hierarchy Analysis (CHA) [33] of the Android framework, including Binder Inter Process Communication (IPC) and Remote Procedure Calls (RPCs). For its initialization, PScout requires the identification of all possible permission enforcement points of the Android framework in order to approximate whether or not a particular API invocation triggers a permission check using a backward reachability traversal approach over the CFG. Though someone may consider PScout's permission map partially outdated, since the latest list available online is for Android version 4.1.1[6], it is worth noting that in several cases it is an essential part of other research works that focus on privacy leaks identification, and malware detection [59, 99]. PScout is based on several assumptions for mapping between protected API methods and permissions that may produces false positives.

Similarly, Bartel *et al.* [26] utilize different static analysis methods to extract the permission map. Specifically, they have generated a CFG based on CHA for Android version 4.0.1 at bytecode level, in several variations (CHA-Naive, CHA-Android and Spark-Android) with the aid of Soot [87] and Spark [56]. Firstly, they performed a string analysis to mine the permission names from the CFG. Then, they carried out a service redirection to pair the service caller to services. Finally, with service identity inversion they removed the unnecessary code from their initial CFG. Although in their research they provide all the necessary evidence that their analysis produces reliable results, they have neither published the code of their method, nor they have provided any experimental verification. Finally, the generated permission map is not any longer available[7]. Additionally, the proposed framework is unable to detect dynamic code loading.

Axplorer [25] relies on the static analysis of the Android framework, as the aforementioned methods do, aiming, however, at a much higher precision. In contrast to other solutions, Axplorer conducts an in-depth static analysis of the Android framework by constructing inter procedural CFGs. In order to improve precision, instead of analyzing the CHA, Axplorer exploits the advantages of object sensitive pointer resolution [73]. It is important to stress that only the public API methods are included in the analysis while hidden API methods are excluded. Furthermore, Axplorer results highlight that the PScout's [24] permission map contains inconsistencies; mainly false positive identifications.

Last but by no means least, another work that focus on Android permission mapping is presented in [29]. A solution (named DPSpec) in which the permission mapping is generated based on XML annotations and Javadoc comments of Android SDK is introduced. Although,

[6]`http://pscout.csl.toronto.edu`

[7]`https://www.abartel.net/permissionmap/`

authors claim higher accuracy compared to Axplorer they do not provide neither the source code of their tool nor the permission map. The proposed tool is limited to construct permission maps from Android's SDK source code annotations, Javadoc comments and XML annotations only.

Solution	**Characteristics**	**Android Version**	**Availability**	**Adaptability**
Stowaway [41]	Dynamic analysis framework based on reflection (method invocation) and modifications of the underlying OS (permission enforcement points monitoring). It requires manual effort to be adaptable to a variety of AOSP versions.	2.2	Not available source code nor permission maps.	No
PScout [24]	Static analysis framework based on CFGs and CHA with several assumptions for mapping between protected API methods and permissions.	2.2.3-5.1.1	Available source code and permission maps.	Unknown
Bartel *et al.* [26]	Static analysis framework based on CFGs and CHA in several variations unable to detect dynamic code loading.	4.0.1	Not available source code nor permission maps.	Unknown
Axplorer [25]	Static analysis framework based on inter procedural CFGs and sensitive pointer resolution designed to analyse only the public API methods.	4.1.1 - 6.0	Not available source code but available permission maps.	Unknown
DPSpec [29]	Static analysis tool able to parse and construct permission maps from Android's SDK source code annotations, Javadoc comments and XML annotations.	Any	Not available source code nor permission maps.	Yes
Dypermin [65]	Dynamic analysis framework based on Android SDK source object correlation map, reflection (method invocation) and permission based Security Exception extraction. Depended on valid object generation.	Any	Available source code but not available permission maps.	Yes

Table 4.1 Method Comparison for Android Permission Mapping Generation

4.4.3 Experimental Evaluation

To assess *Dypermin*'s accuracy, correctness and performance, it has been deployed on two different Android emulators configured with versions 4.1.1 and 6.0 correspondingly, run on an Intel Core i5 1,3 GHz CPU with 8 GB RAM. The selection of the above Android versions was based on the fact that they are considered as the major Android milestones, but also because other related methods, such as Stowaway [41], PScout [24], Axplorer [25] and Bartel *et al.* [26], were tested on the same versions. Furthermore, for the evaluation four managers *i.e.*, `TelephonyManager`, `WiFiManager`, `ConnectivityManager`, and `SmsManager` and two other general purpose classes *i.e.*, `SystemVibrator` and `SipAudio`, have been selected in order to demonstrate the completeness of the approach.

Through the Dypermin's code generation module, described in Subsection 4.4.1, and without loss of generality, the total number of classes, from the Android source code related packages[8], that the *Dypermin* tool could instantiate were counted with static analysis (based on the recursive calculation of class constructors and methods arguments). Specifically, for Android version 6.0, *Dypermin* would be able to generate code for 28170 classes. The total number of methods from classes that *Dypermin* was able to successfully calculate their instantiation in the previews step (static analysis and recursive computation based on their arguments also available in the SDK dump) were 247897 out of 276067 methods. The reason that *Dypermin* could not invoke 10.3% of the methods, is that they belong in abstract classes or that they contain arguments that belong to abstract classes or interfaces or their constructors cannot be invoked as well. The efficiency of *Dypermin* has been based on the static analysis since the compilation time of any generated application is relatively high, a fact already depicted in Figure 4.4 and Figure 4.5.

Since today a complete list of permissions related to protected API methods is not available, the evaluation of the accuracy of the proposed solution, as well as its comparison to other methods, has been solely based on the heuristic formula (1) that provides the full coverage of Android's Framework permissions. That is the set of protected API methods that according to *Dypermin*, other methods, the SDK source and other public documents, are requiring at least one permission.

$$|Dypermin \cup SDKSource \cup PublicDocumentation \cup Other| (1)$$

Note that none of the existing wokrs tries to accomplish such an holistic comparison. Results highlight that *Dypermin* can infer with high accuracy the map between protected API methods and permissions, as it overwhelms both public and SDK documentation (Figure 4.3). Specifically, according to formula (1) for the Android version 4.1.1 the identification rate for permissions - protected API methods, reaches 91.6%, for the `TelephonyManager` class, 94.5% for the `WiFiManager` class, 94.2% for the `ConectivityManager` class, 72.7% for the `SmsManager` class, 100% for the `SystemVibrator` Class and 12.5% for the `SipAudioCal` class.

[8] `com.android.* ,android.*`

Class Name	Android 6.0		
	Dypermin	**SDK Source**	**Documentation**
`TelephonyManager`	91.6%	66.6%	37.5%
`WiFiManager`	94.5%	0%	0.02%
`ConectivityManager`	94.2%	14.2%	22.8%
`SmsManager`	72.7%	27.2%	27.2%
`SystemVibrator`	100%	0%	0%
`SipAudioCall`	12.5%	12.5%	12.5%

Table 4.2 Protected API methods coverage comparison between *Dypermin* and Android's available sources for Android version 4.1.1.

The corresponding results for the Android version 6.0 are 92.7%, 100%, 79.2%, 57.1% 100% and 33.3% respectively. It can be clearly noticed from Table 4.2 and Table 4.3 that in all cases public and SDK source documentation report a smaller number of protected API methods, as compared to *Dypermin*. In other words the currently available documentation is missing important information about the permissions that protected API methods are requiring for their execution, something that, as already explained, *Dypermin* can accurately identify eliminating any false positive results.

Moreover, *Dypermin* in this point it should be mentioned that is able to mine also system permissions, which are those used only by system applications. For instance, consider the case of a non-public API of the `TelephonyManager` class such as `getCompleteVoice-MailNumber(int)` for which *Dypermin* identifies as a required permission for its execution the "`CALL_PRIVILEGED`". Note that this permission is only available to system application, according to Android public documentation[9] and as a result there is not any other way to retrieve this pair of API method - permission beforehand.

Regarding *Dypermin*'s performance evaluation in terms of the time consumed for the generation of the permission map (object generation, application building, installation which is incorporated in the building time and execution), Figures 4.4 and 4.5 illustrate a snapshot

[9] `https://developer.android.com/reference/android/Manifest.permission.html#CALL_PRIVILEGE`

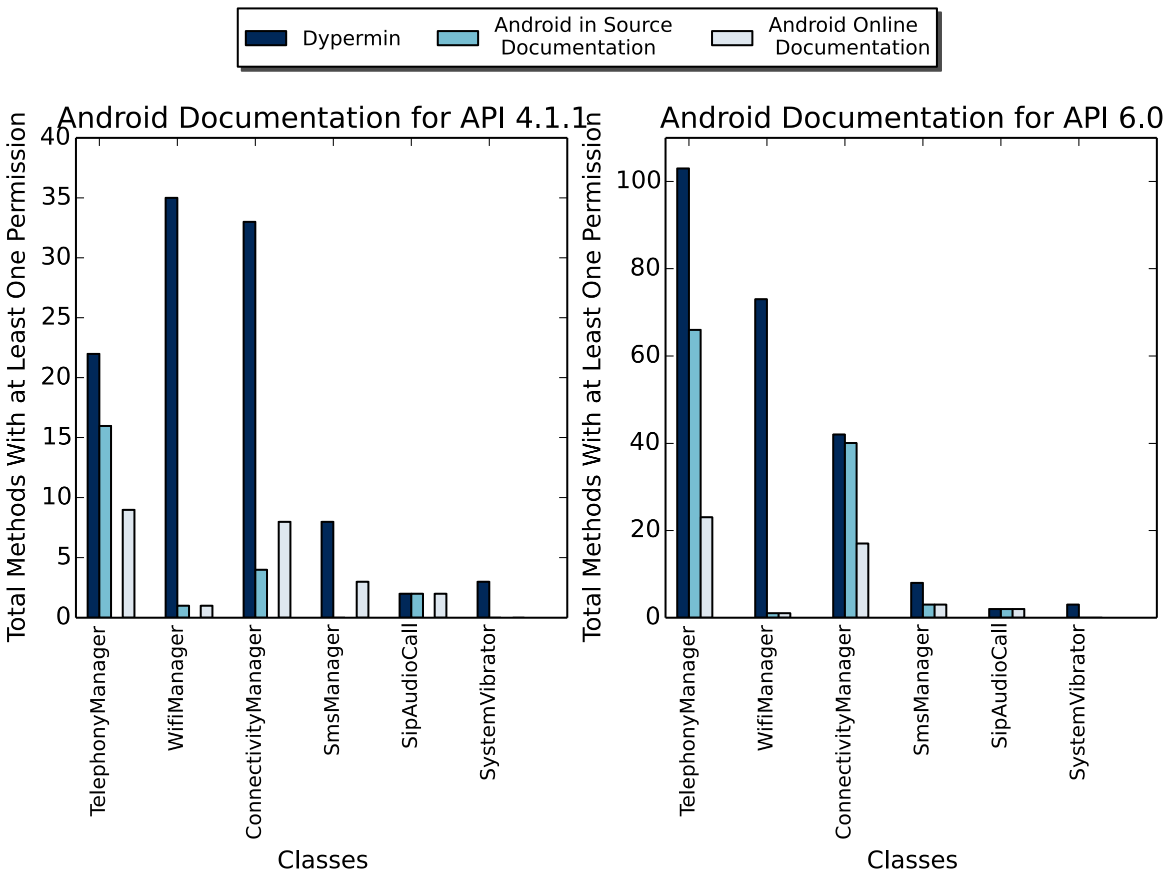


Figure 4.3 Total number of methods with at least one permission as identified by *Dypermin* in comparison with Android software development kit source code and its online counterpart documentation.

of the required processing time for each phase per different method[10]. Results are provided for the different Android versions and for all the classes that were tested. Briefly, outcomes demonstrate that *Dypermin* requires on average less than 17 seconds for code generation, building and testing for each method on both Android 4.1.1 and 6.0. For instance, the worst case scenario was for `TelephonyManager` class in which the code generation, building and testing consume up to 0.06, 13.4, and 3.8 seconds respectively. For the remaining classes that were tested the average processing time is slightly lower. Without loss of generality it can be noticed that the most time consuming phase is that of application's built, while the execution time varies depending on the method. Nevertheless the latter processed time can be considered negligible as it reaches up to 0.06 sec.

[10]Recall that for every method a new application is generated, installed and executed on an emulator.

Class Name	Android 6.0		
	Dypermin	**SDK Source**	**Documentation**
`TelephonyManager`	92.7%	61.1%	21.2%
`WiFiManager`	100%	0.01%	0.01%
`ConectivityManager`	79.2%	75.4%	32%
`SmsManager`	57.1%	21.4%	21.4%
`SystemVibrator`	100%	0%	0%
`SipAudioCall`	33.3%	33.3%	33.3%

Table 4.3 Protected API methods coverage comparison between *Dypermin* and Android's available sources for Android version 6.0.

Overall, *Dypermin* needs less than 17 seconds for generating the permission map for a single method. Eventhough such an overhead per method it can be considered insignificant, the generation of the permission map for the entire Android framework will require a very long processing. To eliminate this significant processing overhead the testing procedure can be parallelized, on different emulators.

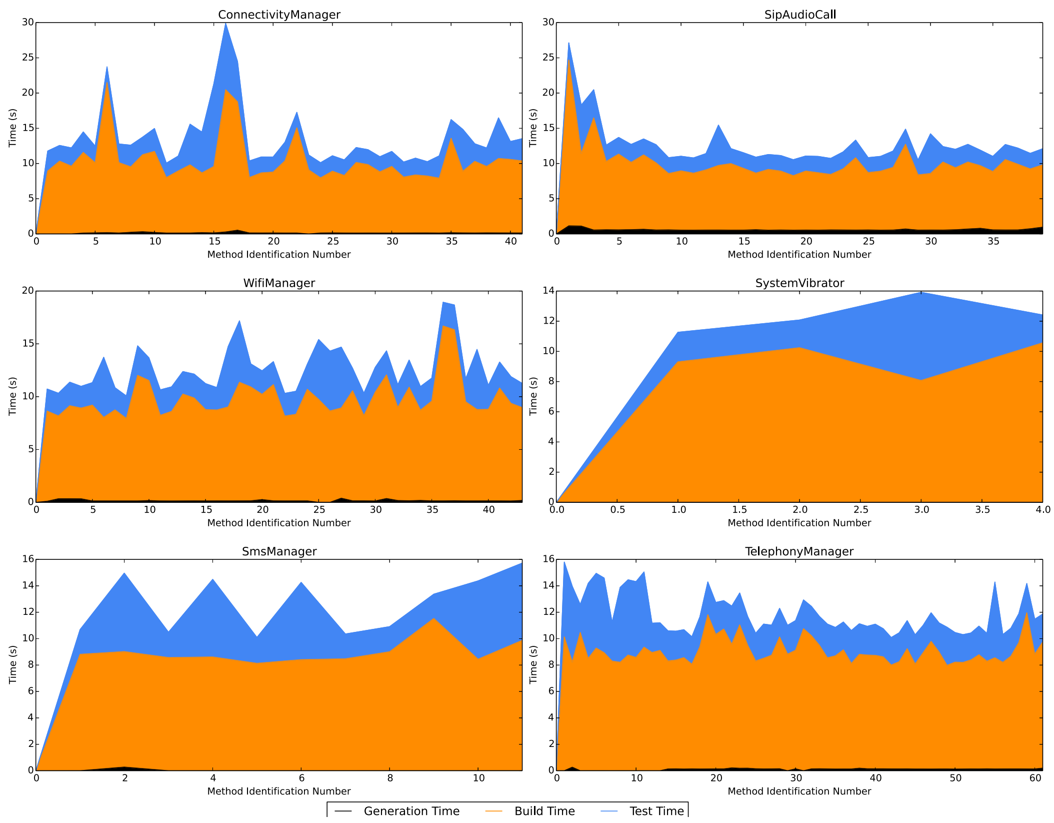


Figure 4.4 A snapshot of *Dypermin*'s time performance per method on Android Version 4.1.1

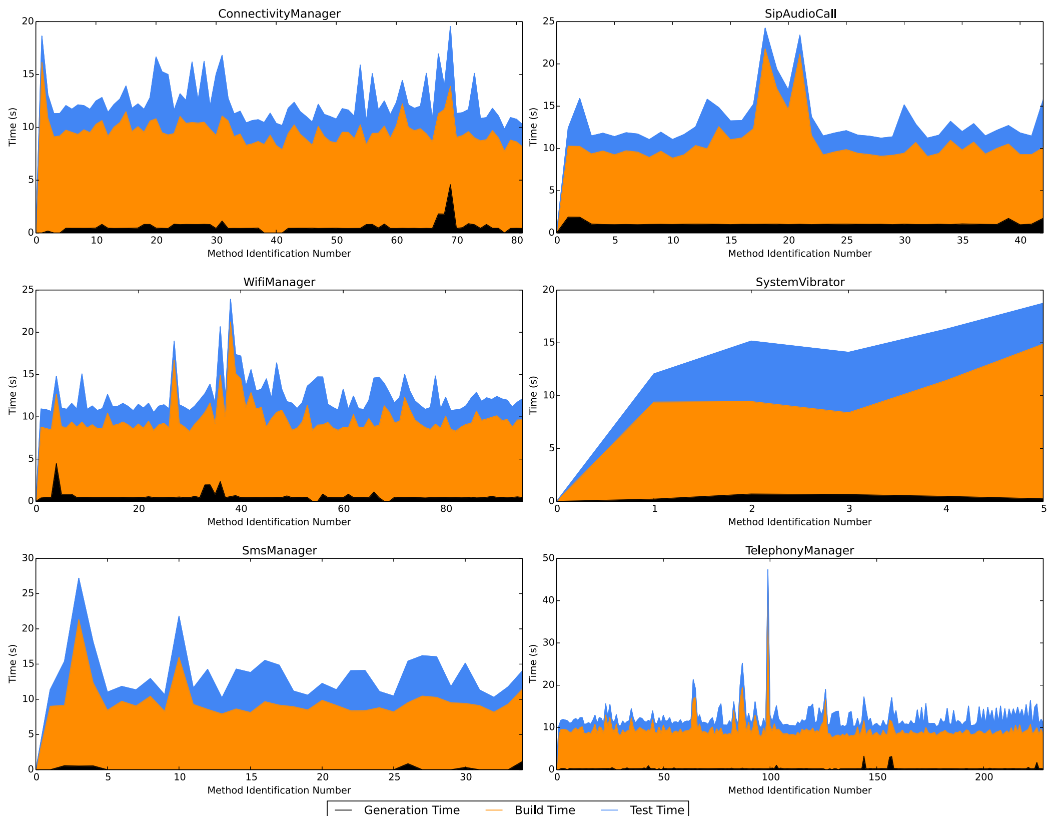


Figure 4.5 A snapshot of *Dypermin*'s time performance per method on Android Version 6.0

Chapter 5

Android Task Management

The Android framework offers the option to activities from different applications to coexist under the same task. This, unlike other OSes, refers to a multitasking mechanism for providing certain functionality *i.e.*, to maintain the state of the activity without launching a new instance or clearing anything on top of it. Though such a mechanism, as already mentioned, offers several advantages *i.e.*, seamless transfer between activities and applications, a malicious entity may try to manipulate it in order to launch hijacking attacks [81, 89, 79, 80, 55, 93, 61, 92]. The comparison of the methods, proposed in the literature, for detecting and preventing the Android task and activity hijacking attacks, can be found in Table 5.1.

5.1 Threats of Android Task Management

The research by Ren *et al.* [79] was the first that systematically study and address the activity hijacking vulnerabilities in Android OS. Attacks against Android's task handling mechanism can be accomplished by manipulating the attributes of `<activity>`[1] elements, like `taskAffinity`, `launchMode`, *etc.*, defined in the target application's manifest and/or intent flags. Activity's attributes, among others, define in which task the activity belongs. By default all app's activities belong to the same task. Whenever the `taskAffinity` attribute points to a different package name from the one to which it belongs, the initiated activity is assigned to a different task affinity. Note that in cases where the `taskAffinity` attribute is combined with the `allowTaskReparenting`, the activity is pushed to the targeted application's task stack; that is defined in the `taskAffinity` attribute.

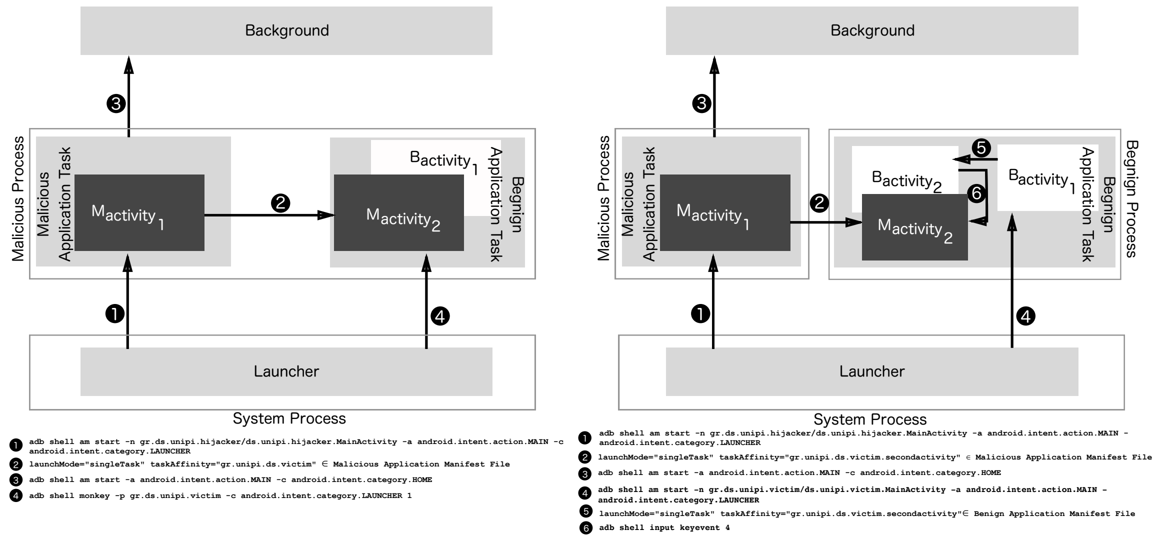
It is evident that a malicious application can interfere with other (benign) applications silently, neither being recognized by end user nor by the underlying OS, by simply ex-

[1] https://developer.android.com/guide/topics/manifest/activity-element

Solution	**Vulnerability Discovery**	**Protection**	**Implementation Layer**	**Prerequisites**	**Availability**
Ren *et al.* [81]	Yes	Exploitation	Application	-	No
Wang *et al.* [89]	Yes	Exploitation	Application	-	No
Ren *et al.* [79]	Yes	Threat Identification	-	-	No
Xiao *et al.* [93]	Yes	Detection	Application	-	No
Ren *et al.* [80]	No	Prevention	Runtime	Device to be rooted	No
Lee *et al.* [55]	No	Detection	Application	-	Yes
Liu *et al.* [61]	Yes	Detection	Application	Bouncer	Yes
Wu *et al.* [92]	No	Prevention	Operating System	OS modification	No
Hwang *et al.* [48]	No	Prevention	Operating System	OS modification	No
TaskAuth [66]	No	Prevention	Operating System	OS modification	Yes
Anactijax [66]	Yes	Detection / Exploitation	Application	-	Yes

Table 5.1 Task and Activity Hijacking Related Works Comparison.

tracting the required information from the targeted application's manifest. In particular, a malicious application can inject an activity into another application's task stack by pointing its `taskAffinity` attribute to the target (legitimate) application's package name and the `launchMode` attribute being equals to `singleTask`, or equivalently the malicious activity is initiated with the intent flag `FLAG_ACTIVITY_NEW_TASK` set. The same result, against a targeted app, can also occur with various combinations of flags and attributes *i.e.*, by using `allowTaskReparenting` attribute that the malicious application 'manipulates' in order to accomplish hijacking attacks. Similarly, benign applications that expose their activities to other applications are vulnerable against the same type of attacks. Figure 5.1 shows how a malicious application can accomplish a hijacking attack against any app on the left side. On the right side, a slightly different case is illustrated where an application explicitly defines a `taskAffinity` for an activity that can be manipulated by malicious applications.



```
❶ adb shell am start -n gr.ds.unipi.hijacker/ds.unipi.hijacker.MainActivity -a android.intent.action.MAIN -c
  android.intent.category.LAUNCHER
❷ launchMode="singleTask" taskAffinity="gr.unipi.ds.victim" ∈ Malicious Application Manifest File
❸ adb shell am start -a android.intent.action.MAIN -c android.intent.category.HOME
❹ adb shell monkey -p gr.ds.unipi.victim -c android.intent.category.LAUNCHER 1
```

```
❶ adb shell am start -n gr.ds.unipi.hijacker/ds.unipi.hijacker.MainActivity -a android.intent.action.MAIN -
  android.intent.category.LAUNCHER
❷ launchMode="singleTask" taskAffinity="gr.unipi.ds.victim.secondactivity" ∈ Malicious Application Manifest File
❸ adb shell am start -a android.intent.action.MAIN -c android.intent.category.HOME
❹ adb shell am start -n gr.ds.unipi.victim/ds.unipi.victim.MainActivity -a android.intent.action.MAIN -
  android.intent.category.LAUNCHER
❺ launchMode="singleTask" taskAffinity="gr.unipi.ds.victim.secondactivity"∈ Benign Application Manifest File
❻ adb shell input keyevent 4
```

Figure 5.1 Hijacking Execution Flows.

5.2 Android Task and Activity Hijacking Attacks

Various researchers have attempted to address task [55, 48, 93, 61, 92] and activity [81, 89, 79, 80] hijacking vulnerabilities, through various different approaches. Currently, some of them have become obsolete [81, 89] due to Android framework's revisions, others require device 'rooting' [80] to enforce controls, others demand end users' intervention [93] assuming that their capable of recognizing malicious activities, others suffers from well-known limitations of static analysis in Android applications (for instance: disregard of implicit intents invocations) [55, 48]. Orthogonal solutions, such as [61], propose the deployment of additional static and dynamic analysis controls at vetting process, whereas authors in [92] introduce a system level mechanism, by modifying the underlying OS, in order to inform end users for possible activity hijacking attacks. In [48] a reformation of the current Android application development framework is required for introducing a signature based mechanism to protect Android applications against task hijacking attacks.

5.2.1 Exploitation and Detection Mechanisms

Ren *et al.* [81] demonstrate a phishing attack against Android activities that require end user's input such as username and password. To do so, the proposed attack assumes that a malicious service is constantly running in the background in order to remain 'hidden'. Whenever the initialization of the target process is detected a malicious fake activity appears, impersonating the original one, by deceiving the end user and steal her/his credentials. In the same direction, Wang *et al.* [89] develop an application named *ActivityHijacker* that mimic legitimate application's activities and perform phishing attacks. Authors demonstrate the applicability of their approach by performing activity phishing attacks against twenty two (22) banking applications, and introduce a static analysis tool, named DroidChain, that can identify malicious application's behaviors based on legitimate application's invocations sequences. However, security flaws identified both by Ren *et al.* [81] and Wang *et al.* [89] are no longer applicable due to system modifications at Android's latest versions.

In another work, Ren *et al.* [79] was the first that systematically study and address the activity hijacking vulnerabilities in Android OS. Their research reveals different types of attacks against activities such as (a) phishing, (b) denial of service, (c) end user monitoring and (d) spoofing. Authors analyze 6.8 million applications to highlight the importance of the identified vulnerabilities in the Android framework and demonstrate that various applications attributes might be manipulated for achieving activity hijacking attacks, and introduce a theoretical mechanism able to circumvent activity hijacking in the Android framework, in which it is recommended that the `taskAffinity` attribute should be scrutinised by Google Bouncer [74].

Lee *et al.* [55] in their work demonstrate how a malicious activity can be injected in a legitimate (target) application's task stack. To do so, a malicious application should have define the target application's package name to the `taskAffinity` attribute in its manifest file, while the corresponding malicious activity is "initiated" as a new task with the intent flag `FLAG_ACTIVITY_NEW_TASK` set. Authors considering the different combinations of `launchMode` and task related intent flags demonstrate that such an attack can be performed in more than two hundred ways, and deploy a static analysis tool named AIDetector[2] to detect vulnerable applications by correlating information extracted from application's manifest and its disassembled code. The AIDetector reports possible injection cases whenever an application declares activities that contain different `taskAffinity` attribute from their (original) package name, combined with attributes such as the `lunchMode`. However, AIDetector does not consider implicit intents or flows involving built-in method calls or static fields, and might report false positives combinations as it does not trigger the execution of the activity. Authors extend their approach in [48] where they introduce a signature-based defense mechanism that requires the modification of: (a) Android application's structure (by adding new tags into activity attributes), (b) Android framework, and Google's vetting process to enforce their scheme. More precisely, authors introduce a signature-based activity access control mechanism based on policies that define each application's interactions with other services and applications. The policies are defined by application developers during the development

[2] `https://github.com/SunghoLee/AIDetector`

phase. So, whenever a developer needs to modify an application interaction policy, all the relevant applications should be recompiled and republished to the Google Play store.

Xiao *et al.* [93] study activity injection attacks by combining, for both the malicious and benign activities, (a) the `allowTaskReparenting`, (b) the `taskAffinity` attributes (c) the different launch Intent flags and (d) the sequences of launching events. They developed an activity hijacking vulnerability tool that follows a similar approach with *Anactijax*. It combines activity attributes and intent flags to generate pairs of malicious and benign applications that are executed under various scenarios to identify possible task interference combinations. In this way, the authors demonstrate four task interference proof-of-concept attacks: (a) UI Phishing, (b) Activity-in-the-middle, (c) Gallery Stealing and (d) Screen Shot Capturing. In order to minimise the attack surface and protect end users against such a threat, the authors introduce a task interference detection application named Task Interference Checking (TICK), to detect potential task interferences among applications. Although, TICK has no prevention capabilities. In fact, TICK scans either an application before installation or periodically a set of installed application to report possible activity hijacking vulnerabilities, without allowing any action to be taken against malicious activities.

Liu *et al.* [61] in their study propose a tool, named TDroid[3], applicable to apps' distribution market in order to enhance the vetting [74] process. TDroid combines static and dynamic analysis to detect malicious applications that attempt to replace legitimate application's top activity. During the static analysis phase, TDroid transforms the application into runnable slices to check if they contain potentially activity related attacks. In the second phase of dynamic analysis, TDroid executes these slices on an Android phone or emulator and extracts possible malicious activities. However, such an approach does not consider: (a) the inter-component communication (ICC) and reflection APIs, (b) the data dependencies through persistent storage that might result in partial hijacking attacks identification.

Lyvas *et al.* [66] developed a tool, named *Anactijax* (Android Activity Hijacking Examiner), capable of accomplishing activity hijacking attacks against other (target) applications, demonstrating through experiments how a malicious application functions in such a scenarios. *Anactijax* provides a framework for automatically generating a malicious application to accomplish activity hijacking attacks. This is achieved by retrofiring the malicious application generator with specific flags and attributes *i.e.*, such info can be extracted from legitimate (target) application's manifest..

On one hand, *Anactijax*, similar to Xiao *et al.* [93], automatically calculates all the combinations of intent flags and `<activity>` attributes extracted from the Android Software Development Kit (SDK), aiming at generating malicious activities that can be injected against

[3]https://tdroidtool.github.io

any benign application, employing a brute force approach. On the other hand, *Anactijax* has the capacity to generate activity injection use cases against real-world Android applications; a functionality that, no other current tool supports. As Figure 5.2 depicts, *Anactijax* for each combination of intent flag and `<activity>` attribute, *Anactijax* deploys a malicious application against a benign one (target), and deduce whether the malicious activity has been successfully pushed in the benign application's task by monitoring system's and application's logs.

To demonstrate activity hijacking attacks an environment where a malicious and a benign application co-exist in an Android device, has been assumed. The malicious application runs in the background and launches attacks against any given (target) application considering the default settings *i.e.*, the target application exports the main activity. In order to assess its accuracy and correctness, *Anactijax* has been deployed on a standard Android emulator version 9 with 2 GB of RAM and 1 Central Processing Unit (CPU), running on a machine with an Intel Core i5 1,3 GHz CPU and 8 GB RAM.

For testing purposes a customized benign application was developed, that contains a main activity which is by default exported and, alternatively, that it declares an explicit `taskAffinity` for an activity that *Anactijax* targets. In this context, *Anactijax* generates all the possible combinations considering different activity attributes with related intent flags. For instance, considering the use of 2 or 3 activity attributes with related intent flags, *Anactijax* creates 3744 uses cases to be executed against the target app, and reports 80 successful activity injection attacks. Table 5.2 provides and overview of the analysis of the tested use cases. Results suggest that successful hijacking attacks combine either activity's `taskAffinity` and `launchMode` attributes, or rely on `FLAG_ACTIVITY_NEW_TASK` intent flag.

The ability of *Anactijax* to produce malicious applications which can launch activity injection attacks against real-world applications, relies on its capacity to extract the appropriate features from the targeted APK as Figure 5.3 depicts. For any given real-world application, *Anactijax* extracts its manifest file, exports its package name, the main activity's name, and any other activity name with the `taskAffinity` value set. Note that the information in the manifest file is in clear text as it is used by Android OS to execute properly the application. The overall approach has been already described above.

In particular to demonstrate that a malicious application can take the control of a benign (target) one: case (a) (left side of Figure 5.1) during the initialization of the target application, *Anactijax* generates a hijacking application that contains a manifest with a `taskAffinity` attribute value equal to the target application's package name and adds the necessary Java code for the initialization of the malicious activity with the intent flag `FLAG_ACTIVITY_NEW_-TASK` set in the malicious activity's source code, and afterwards it generates the malicious

APK and case (b) (right side of Figure 5.1) while the user navigates to benign application activities *Anactijax* injects a malicious activity into the benign application in such a way that the malicious task is placed in the stack of the benign application and will be triggered when the user navigates back. To do so *Anactijax* searches if the given benign application code activities define explicitly the `taskAffinity` attribute. If this is the case it extracts the `taskAffinity` value and generates a hijacking application that contains a malicious activity with the `taskAffinity` attribute value equal to the victim activity's one (in the malicious applications Manifest file). Then it adds the necessary Java code for the initialization of the malicious activity with the intent flag `FLAG_ACTIVITY_NEW_TASK` set in the malicious activity's source code in order to be capable to be injected into target application's stack.

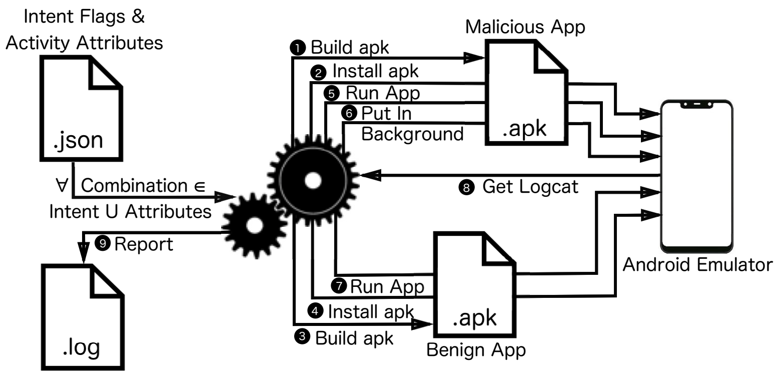


Figure 5.2 High-Level Design of *Anactijax* for Task and Activity Injection Related Activity Attributes and Intent Flag Exfiltration.

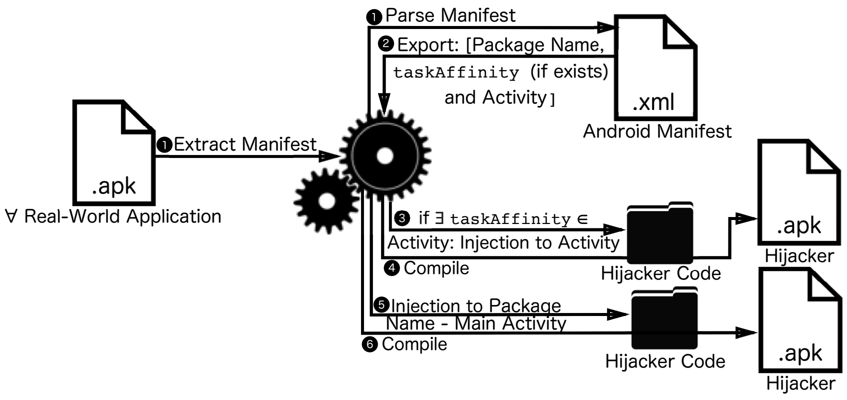

❶ Parse Manifest
❷ Export: [Package Name, taskAffinity (if exists) and Activity]
.xml
Android Manifest
.apk
∀ Real-World Application
❶ Extract Manifest
❸ if ∃ taskAffinity ∈ Activity: Injection to Activity
❹ Compile
Hijacker Code
.apk
Hijacker
❺ Injection to Package Name - Main Activity
❻ Compile
Hijacker Code
.apk
Hijacker

Figure 5.3 High-Level Design of *Anactijax* for Automated Hijacking Application Generation Against Real-World Applications.

5.2.2 Prevention Mechanisms

Wu *et al.* [92] provided a detection service that alerts end users for possible password phishing attempts via activity hijacking attacks. The proposed system-level mechanism is limited only to inform users via pop-up windows for potential activity hijacking attacks whenever password fields are involved.

In an extended approach of Xiao *et al.* [55] a signature-based defense mechanism named SAAC was proposed in [48] that requires the modification of: (a) Android application's structure (by adding new tags into activity attributes), (b) Android framework, and Google's vetting process to enforce their scheme. More precisely, authors introduce a signature-based activity access control mechanism based on policies that define each application's interactions with other services and applications. The policies are defined by application developers during the development phase. So, whenever a developer needs to modify an application interaction policy, all the relevant applications should be recompiled and republished to the Google Play store.

Ren *et al.* [80] designed an application able to prevent Graphic User Interface (GUI) attacks against Android named WindowGuard, a solution that requires the device to be 'rooted' as it is deployed upon Xposed framework [34]. WindowGuard enforces an android window integrity mechanism (AWI) that provides the following enhancements into Android GUI system: (a) an activity session integrity, (b) access control for the free windows, and (c) safeguarding of focused activity transition. The effectiveness of WindowGuard has been evaluated over 12,060 applications, which 1% trigger warning for possible activity manipulation.

In order to prevent task and activities hijacking attacks and, at the same time, minimize Android's attack surface, it is proposed [66] to enhance the Android's task management mechanism by introducing a system level service, namely the *TaskAuth*, that authenticates the activity's source whenever it invokes other applications. In order for the proposed approach to be transparent, it leverages on application's signature verification scheme, used during application's initialization, by extending the `ActivityManager` system service. In this way, *TaskAuth*, does not require any intervention or application modification by end users or developers respectively.

More specifically, *TaskAuth*, verifies whether or not the identities of source and target activities match, as depicted in Figure 5.4. To do so, *TaskAuth* whenever an activity (source) invokes another one (target), *TaskAuth* scrutinizes source activity's `taskAffinity` and `processName` fields of `aInfo` object[4] and checks if the source activity declares another application's package name in the `taskAffinity` field. If this is not the case, *TaskAuth* does not intervene and the execution is processed normally, otherwise it retrieves a PackageInfo

[4]belongs to the `ActivityInfo` class.

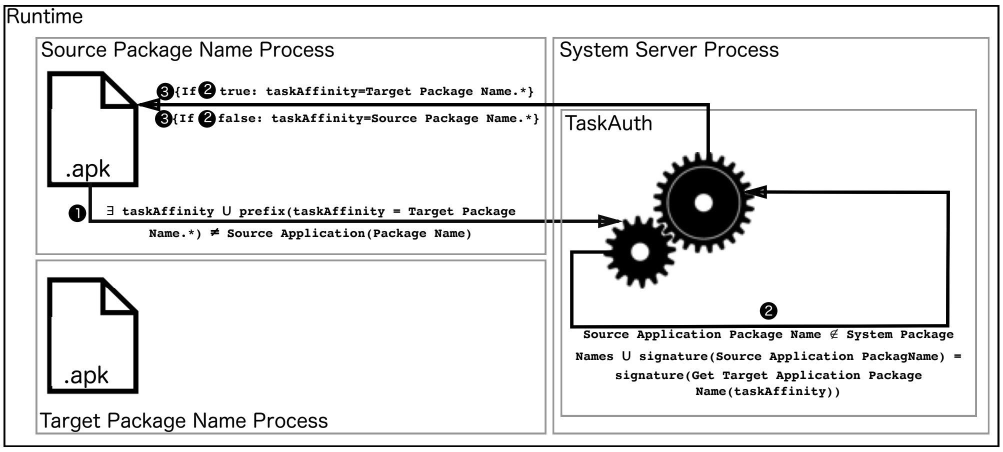

Runtime
Source Package Name Process
System Server Process
❸{If ❷ true: taskAffinity=Target Package Name.*}
❸{If ❷ false: taskAffinity=Source Package Name.*}
TaskAuth
.apk
❶ ∃ taskAffinity ∪ prefix(taskAffinity = Target Package Name.*) ≠ Source Application(Package Name)
❷
Source Application Package Name ∉ System Package Names ∪ signature(Source Application PackagName) = signature(Get Target Application Package Name(taskAffinity))
.apk
Target Package Name Process

Figure 5.4 *TaskAuth* High Level Design.

object for both activities, by executing the `getPackageInfo(...)`, that among others contains the application's singing certificates and compares them to determine whether source and target activities packages belong to the same developer.

At this point it should be noted that *TaskAuth* considers legitimate the declaration of the `taskAffinity` pointing in another application's package name or in another application's package name, if both belong to the same developer, or the source activity belongs to a trusted organisation *i.e.*, Google. Otherwise, *TaskAuth* classifies it as a task hijacking attempt and 'blocks' it by setting dynamically the `taskAffinity` of the malicious activity to the default value that is its package name. In this way, `taskAffinity` does not affect the application's execution at all. Of course, other policies could be retrofitted to the *TaskAuth* for covering additional needs.

5.2.3 Empirical Analysis

For evaluating [66] the capacity of *Anactijax* to generate malicious applications that can attack real-world applications as well as the ability of *TaskAuth* to prevent these attacks, 300 applications, were randomly downloaded after random sampling from a pull of 5.350 unique application titles crawled from Google Play[5] covering the majority of all the provided application categories. From this initial application set, 52 were supporting only ARM architectures and another 34 applications could not be executed on the emulated environment and thus were excluded. Finally for 14 applications *Anactijax* was unable to extract features.

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

For the remaining 200 applications (three applications on average from 43 different application categories such as games, social, finance, business *etc.*), we used an automated procedure in order to evaluate the functionality of *Anactijax* and the effectiveness of *TaskAuth*. More precisely, *Anactijax* was used to generate hijacking applications that target a specific real-world application. Then, with the aid of a standard Android emulator, we automatically executed the benign and the malicious applications, and by monitoring the results of `dumpsys`[6] we were able to determine if the malicious application managed to inject the malicious activity into a benign application task or if the benign application task was injected into the malicious process.

For the 200 applications, *Anactijax* was able to generate 202 applications (requiring on average, 7.2 s for each one) from which 200 were targeting the given application's main activity with `taskAffinity` pointing to the targeted application's package name (benign task injection into the malicious process - case - a) and 2 malicious applications targeting benign activities with declared `taskAffinity` (malicious activity injection into the benign process - case - b).

After executing each malicious application against each targeted benign application, we identified two cases where the activity injection failed because the developers of the benign applications had defined their main activities with the flag `launchMode` set to a `singleTask` and their back button is used to exit directly the applications rather than navigating among activities. For the rest 198 cases (99%), the malicious applications were able to accomplish successfully activity injection attacks.

At this point it is worth mentioning that 28 targeted applications had declared their main activity with the flag `launchMode` set to `singleTop` something that made them vulnerable to malicious activity injection into the benign process (case - b) rather than being vulnerable to benign task injection into the malicious process (case - a) which was the case for the remaining 170 applications.Finally, for 10 applications out of 28, the malicious activity was injected on top of a splash screen main activity that instantly initiated other activities, a fact that caused the injected activity to remain invisible to the user.

TaskAuth was deployed on a modified Android framework version 9 on the standard Android emulator with 2 GB of RAM and 1 CPU on a machine featuring Intel Core i5 1,3 GHz CPU and 8 GB RAM, and we have evaluated its effectiveness in terms of: (a) efficacy to detect hijacking attacks and (b) overhead introduced *i.e.*,. how *TaskAuth* influences application performance.

[6]The `adb shell dumpsys activity` command shows the current state of running activities into a system along with their task records.

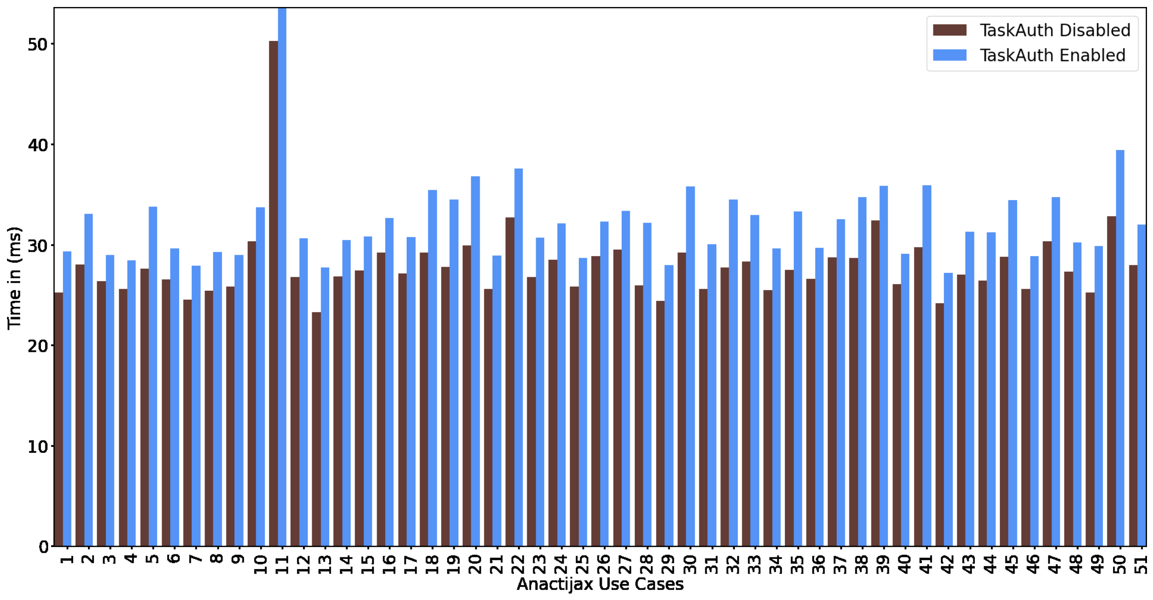


Figure 5.5 Overhead Introduced by *TaskAuth* to the (Default Exported) Main Activity With Default (Application Package Name) `taskAffinity`.

From evaluation standpoint, *TaskAuth* was tested against *Anactijax* were *Anactijax* has not reported any successful injection attack (from the initial 80 injection attacks) when *TaskAuth* is enforced. Table 5.3 overviews the results of the tests, where the malicious application ($M_{process}$) was unable to inject a malicious activity ($M_{activity}$) into the benign task (B_{task} or $B_{process}$), thus allowing the benign process ($B_{process}$) to proceed with its normal execution in all identified activity injection cases.

As far as the *TaskAuth* processing overhead is concerned, results have indicated that it has caused a negligible increase of the execution time *i.e.*, the maximum observed execution time was as little as 7 ms, with a minimum of 2.6 ms, while it requires on average 4.3 ms to detect whether an activity is malicious or not as Figure 5.5 and 5.6 show.

In the same direction, to assess *TaskAuth* efficacy to protect also real-world applications, the *Anactijax* malicious generated applications were reused and repeated the hijacking attacks while the *TaskAuth* was in place. The results demonstrate that *TaskAuth* managed to prevent all the hijacking attack attempts and also that its performance, in terms of the overhead introduced, is similar to the demo use case (see previous paragraphs). In particular, the overhead introduced is on average 4.8 ms, with a minimum of 2.6 ms and a maximum of 6.9 ms as Figure 5.7 illustrates.

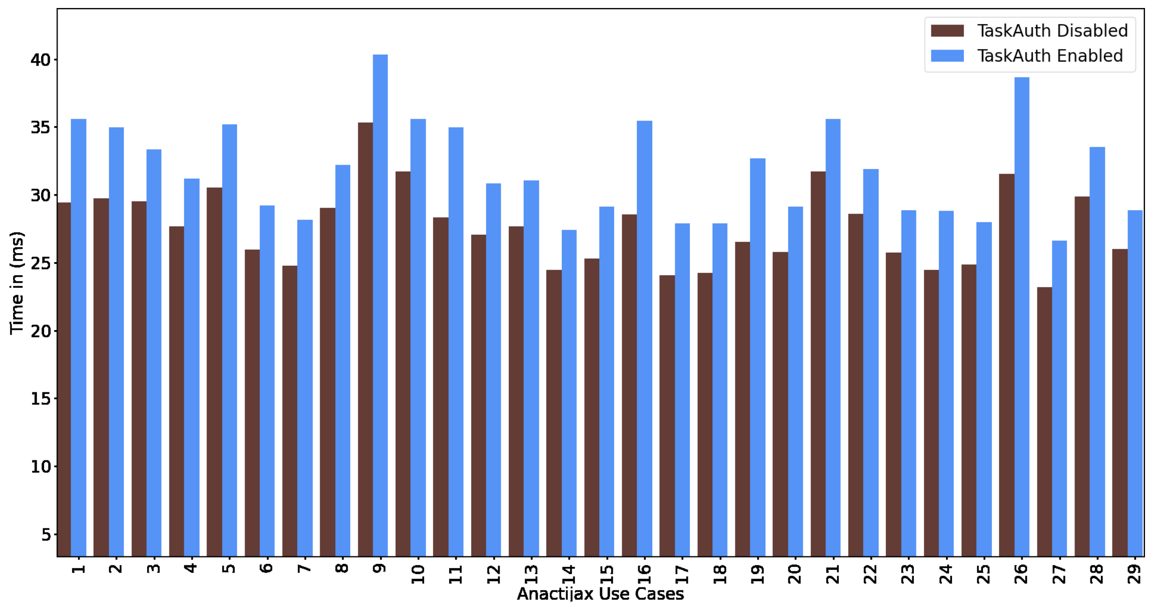


Figure 5.6 Overhead Introduced by *TaskAuth* for Vulnerable Applications With Activities That Declare Explicitly `taskAffinity` Other Than the Default (Application Package Name).

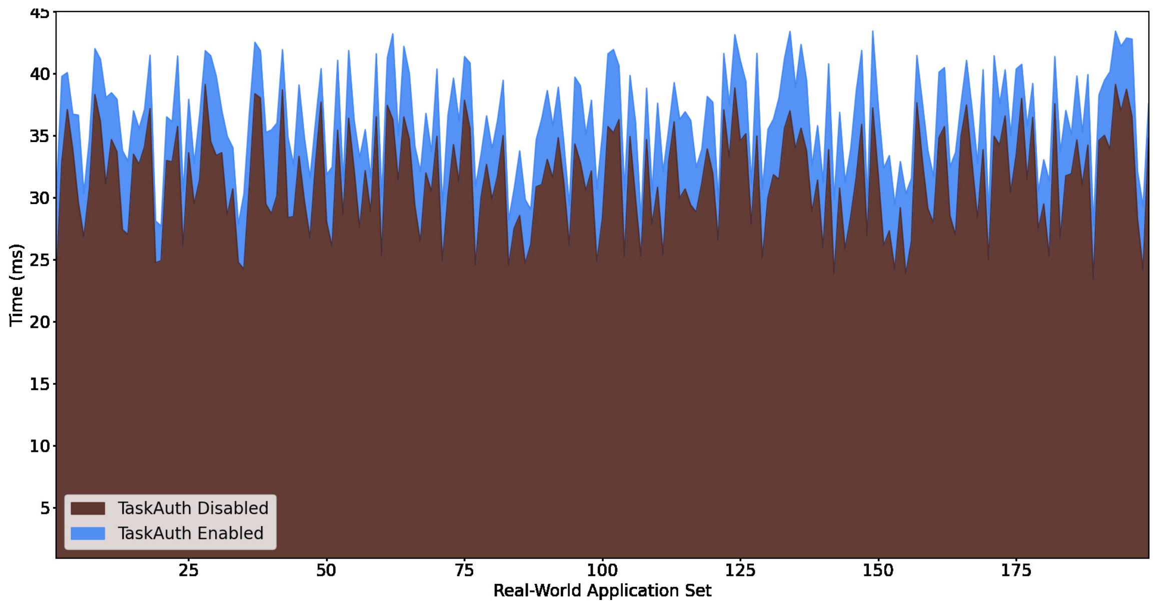


Figure 5.7 Overhead Introduced by *TaskAuth* For Each Real-World Application Used in the Evaluation (Total Two Hundred Applications).

#	Malicious Application		Benign Application		Impact
	Activity Attributes	Intent Flags	Activity Attributes	Intent Flags	
1	`launchMode="singleTask" taskAffinity= "gr.unipi.ds.victim"`	-	-	-	$M_{activity} \in B_{task} \mid B_{task} \in M_{process}$
2	`launchMode="singleTask" taskAffinity= "gr.unipi.ds.victim allowTaskReparenting="true"`	-	-	-	$M_{activity} \in B_{task} \mid B_{task} \in M_{process}$
3	`taskAffinity= "gr.unipi.ds.victim`	`FLAG_ACTIVITY_NEW_TASK`	-	-	$M_{activity} \in B_{task} \mid B_{task} \in M_{process}$
4	`allowTaskReparenting="true" taskAffinity= "gr.unipi.ds.victim`	`FLAG_ACTIVITY_NEW_TASK`	-	-	$M_{activity} \in B_{task} \mid B_{task} \in M_{process}$
5	`allowTaskReparenting="true" taskAffinity= "gr.unipi.ds.victim .secondactivity"`	`FLAG_ACTIVITY_NEW_TASK`	`taskAffinity= "gr.unipi.ds.victim .secondactivity"`	`FLAG_ACTIVITY_NEW_TASK`	$M_{activity} \in B_{task} \mid B_{task} \in B_{process}$
6	`taskAffinity= "gr.unipi.ds.victim .secondactivity"`	`FLAG_ACTIVITY_NEW_TASK`	`taskAffinity= "gr.unipi.ds.victim .secondactivity"`	`FLAG_ACTIVITY_NEW_TASK`	$M_{activity} \in B_{task} \mid B_{task} \in B_{process}$
7	`launchMode="singleTask" taskAffinity= "gr.unipi.ds.victim .secondactivity"`	-	`taskAffinity= "gr.unipi.ds.victim .secondactivity"`	`FLAG_ACTIVITY_NEW_TASK`	$M_{activity} \in B_{task} \mid B_{task} \in B_{process}$
8	`launchMode="singleTask" taskAffinity= "gr.unipi.ds.victim .secondactivity" allowTaskReparenting="true"`	-	`taskAffinity= "gr.unipi.ds.victim .secondactivity"`	`FLAG_ACTIVITY_NEW_TASK`	$M_{activity} \in B_{task} \mid B_{task} \in B_{process}$
9	`allowTaskReparenting="true" taskAffinity= "gr.unipi.ds.victim .secondactivity"`	`FLAG_ACTIVITY_NEW_TASK`	`launchMode="singleTask" taskAffinity= "gr.unipi.ds.victim .secondactivity"`	-	$M_{activity} \in B_{task} \mid B_{task} \in B_{process}$
10	`taskAffinity= "gr.unipi.ds.victim .secondactivity"`	`FLAG_ACTIVITY_NEW_TASK`	`launchMode="singleTask" taskAffinity= "gr.unipi.ds.victim .secondactivity"`	-	$M_{activity} \in B_{task} \mid B_{task} \in B_{process}$
11	`launchMode="singleTask" taskAffinity= "gr.unipi.ds.victim .secondactivity"`	-	`launchMode="singleTask" taskAffinity= "gr.unipi.ds.victim .secondactivity"`	-	$M_{activity} \in B_{task} \mid B_{task} \in B_{process}$
12	`launchMode="singleTask" taskAffinity= "gr.unipi.ds.victim .secondactivity" allowTaskReparenting="true"`	-	`launchMode="singleTask" taskAffinity= "gr.unipi.ds.victim .secondactivity"`	-	$M_{activity} \in B_{task} \mid B_{task} \in B_{process}$

Table 5.2 *Anactijax* Has Successfully Identified Hijacking Combinations of Activity Attributes and Intent Flags.

#	Malicious Application		Benign Application		Impact
	Activity Attributes	Intent Flags	Activity Attributes	Intent Flags	

1	`launchMode="singleTask"`
	`taskAffinity=`
	`"gr.unipi.ds.victim"`
		-	-	-	$M_{activity} \in M_{task} \in M_{process}$,
					$B_{activity} \in B_{task} \in B_{process}$

2	`launchMode="singleTask"`
	`taskAffinity=`
	`"gr.unipi.ds.victim`
	`allowTaskReparenting="true"`
		-	-	-	$M_{activity} \in M_{task} \in M_{process}$,
					$B_{activity} \in B_{task} \in B_{process}$

3	`taskAffinity=`
	`"gr.unipi.ds.victim`
		`FLAG_ACTIVITY_NEW_TASK`	-	-	$M_{activity} \in M_{task} \in M_{process}$,
					$B_{activity} \in B_{task} \in B_{process}$

4	`allowTaskReparenting="true"`
	`taskAffinity=`
	`"gr.unipi.ds.victim`
		`FLAG_ACTIVITY_NEW_TASK`	-	-	$M_{activity} \in M_{task} \in M_{process}$,
					$B_{activity} \in B_{task} \in B_{process}$

5	`allowTaskReparenting="true"`
	`taskAffinity=`
	`"gr.unipi.ds.victim`
	`.secondactivity"`
		`FLAG_ACTIVITY_NEW_TASK`	`taskAffinity=`
			`"gr.unipi.ds.victim`
			`.secondactivity"`
				`FLAG_ACTIVITY_NEW_TASK`	$M_{activity} \in M_{task} \in M_{process}$,
					$B_{activity} \in B_{task} \in B_{process}$

6	`taskAffinity=`
	`"gr.unipi.ds.victim`
	`.secondactivity"`
		`FLAG_ACTIVITY_NEW_TASK`	`taskAffinity=`
			`"gr.unipi.ds.victim`
			`.secondactivity"`
				`FLAG_ACTIVITY_NEW_TASK`	$M_{activity} \in M_{task} \in M_{process}$,
					$B_{activity} \in B_{task} \in B_{process}$

7	`launchMode="singleTask"`
	`taskAffinity=`
	`"gr.unipi.ds.victim`
	`.secondactivity"`
		-	`taskAffinity=`
			`"gr.unipi.ds.victim`
			`.secondactivity"`
				`FLAG_ACTIVITY_NEW_TASK`	$M_{activity} \in M_{task} \in M_{process}$,
					$B_{activity} \in B_{task} \in B_{process}$

8	`launchMode="singleTask"`
	`taskAffinity=`
	`"gr.unipi.ds.victim`
	`.secondactivity"`
	`allowTaskReparenting="true"`
		-	`taskAffinity=`
			`"gr.unipi.ds.victim`
			`.secondactivity"`
				`FLAG_ACTIVITY_NEW_TASK`	$M_{activity} \in M_{task} \in M_{process}$,
					$B_{activity} \in B_{task} \in B_{process}$

9	`allowTaskReparenting="true"`
	`taskAffinity=`
	`"gr.unipi.ds.victim`
	`.secondactivity"`
		`FLAG_ACTIVITY_NEW_TASK`	`launchMode="singleTask"`
			`taskAffinity=`
			`"gr.unipi.ds.victim`
			`.secondactivity"`
				-	$M_{activity} \in M_{task} \in M_{process}$,
					$B_{activity} \in B_{task} \in B_{process}$

10	`taskAffinity=`
	`"gr.unipi.ds.victim`
	`.secondactivity"`
		`FLAG_ACTIVITY_NEW_TASK`	`launchMode="singleTask"`
			`taskAffinity=`
			`"gr.unipi.ds.victim`
			`.secondactivity"`
				-	$M_{activity} \in M_{task} \in M_{process}$,
					$B_{activity} \in B_{task} \in B_{process}$

11	`launchMode="singleTask"`
	`taskAffinity=`
	`"gr.unipi.ds.victim`
	`.secondactivity"`
		-	`launchMode="singleTask"`
			`taskAffinity=`
			`"gr.unipi.ds.victim`
			`.secondactivity"`
				-	$M_{activity} \in M_{task} \in M_{process}$,
					$B_{activity} \in B_{task} \in B_{process}$

12	`launchMode="singleTask"`
	`taskAffinity=`
	`"gr.unipi.ds.victim`
	`.secondactivity"`
	`allowTaskReparenting="true"`
		-	`launchMode="singleTask"`
			`taskAffinity=`
			`"gr.unipi.ds.victim`
			`.secondactivity"`
				-	$M_{activity} \in M_{task} \in M_{process}$,
					$B_{activity} \in B_{task} \in B_{process}$

Table 5.3 Execution Prevention of the Identified Activity Hijacking Use Cases of *Anactijax* after the Enforcement of *TaskAuth*.

Chapter 6

Android Intent Handling

The Android framework offers to application components (from the same or different process) the option of interaction (send/receive or exchange data) via intents which are at a higher-level of abstraction executed over the Binder (as already mentioned in Subsection 3.1.3). Figure 6.1 depicts the control flow of an activity to activity communication between different applications through implicit intents. In detail, in the case where an application requests from some other component to take an action (*i.e.*, open a document, through some available user application) it employs an implicit intent call, as it is not aware which application can handle such a request. Afterwards, the execution flow is transferred to the activity task manager service (ATMS) which is responsible for delivering it to the application that is able to handle the implicit intent request; such applications are those that contain the corresponding intent-filter, and normally the user is the one selecting it. Finally, ATMS delivers the implicit intent to the application (selected by the user) through the Android framework.

6.1 Threats of Android Inter-Process Communication

Inter-Process Communication (IPC) or Inter-Component Communication (ICC) in Android platform is performed over Intent message objects. Activity's implicit intents are used whenever a new activity is initialized with a 'description' of an action (A.2) rather than defining explicitly (A.4) a component to handle it, while simultaneously might be used to exchange data between different processes. This exchange mechanism might arises security and privacy implications since not only the source activity is unaware of the component that will manage it, but also the destination activity does not know the origin of the received intent driving either to *intent redirection* or *malicious activity launch* and *intent hijacking* attacks. On one side, the former category concerns cases where legitimate applications' publicly available exported components (*i.e.*, activities) are exploited by malicious one to send malevolent data

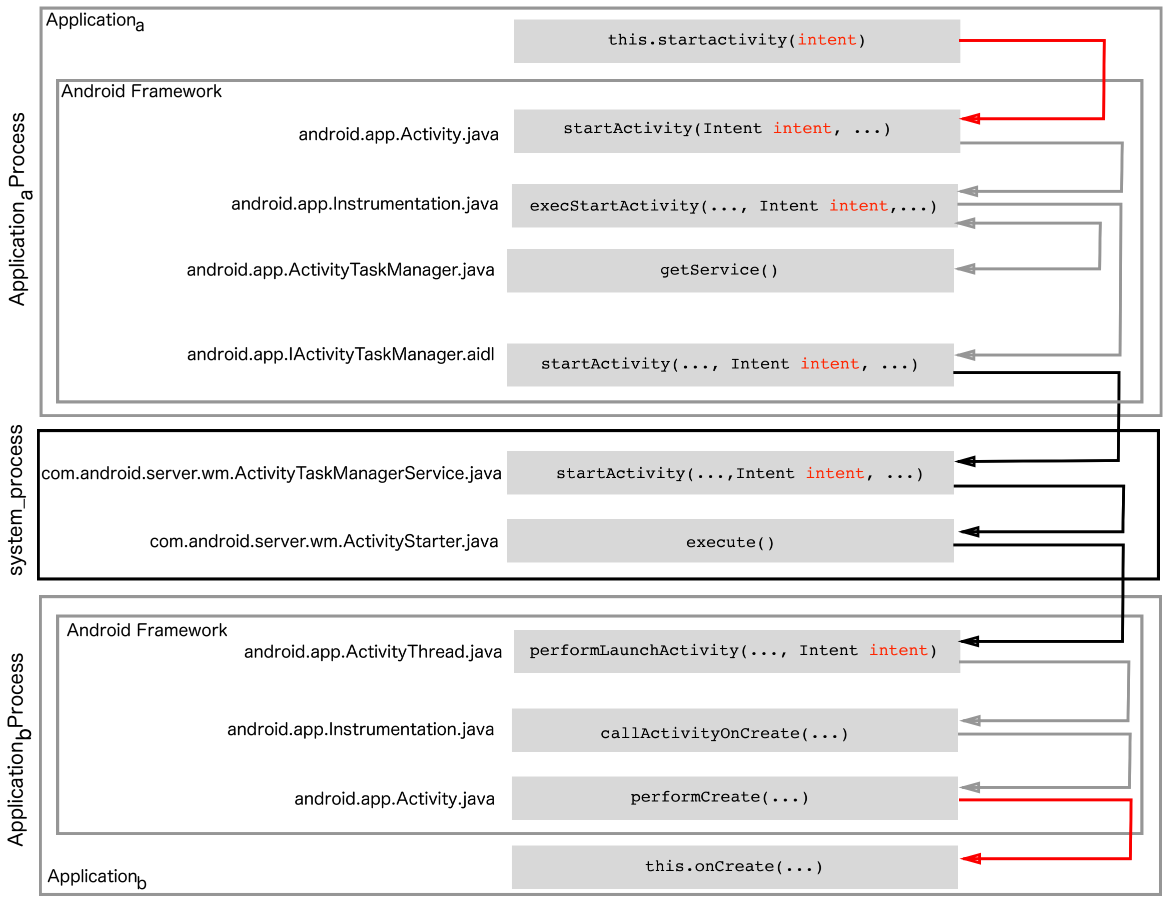

Application$_a$
this.startactivity(intent)
Android Framework
android.app.Activity.java
startActivity(Intent intent, ...)
android.app.Instrumentation.java
execStartActivity(..., Intent intent,...)
android.app.ActivityTaskManager.java
getService()
android.app.IActivityTaskManager.aidl
startActivity(..., Intent intent, ...)
Application$_a$Process
system_process
com.android.server.wm.ActivityTaskManagerService.java
startActivity(...,Intent intent, ...)
com.android.server.wm.ActivityStarter.java
execute()
Android Framework
android.app.ActivityThread.java
performLaunchActivity(..., Intent intent)
android.app.Instrumentation.java
callActivityOnCreate(...)
android.app.Activity.java
performCreate(...)
Application$_b$
this.onCreate(...)
Application$_b$Process

Figure 6.1 Android (Version 11) Activity Handling.

to other legitimate components and services, which might perform sensitive operations upon them, considering the data as trusted due to the fact that a well known source sent them. On the other side, the latest threat is related to malicious applications that define intent filters to intercept a legitimate application's transmitted data and gain unauthorized access to otherwise private data.

Currently, various research works attempt to address these security flaws using different approaches ranging from (a) static analysis detection tools [85, 37] to (b) system-level enhancements to prevent such attacks [88, 95] and minimize the OS's attack surface. Though their effectiveness is unquestionable, some of them suffer [85, 37] from the common limitations of static analysis approaches, while others that require the modification of the underlying OS, cannot be re-configured [88] dynamically, or expose intents beyond the interaction processes [95].

The comparison among the proposed methods regarding the Android *intent redirection* or *malicious activity launch and intent hijacking*, detection, and prevention mechanisms is listed in Table 6.1

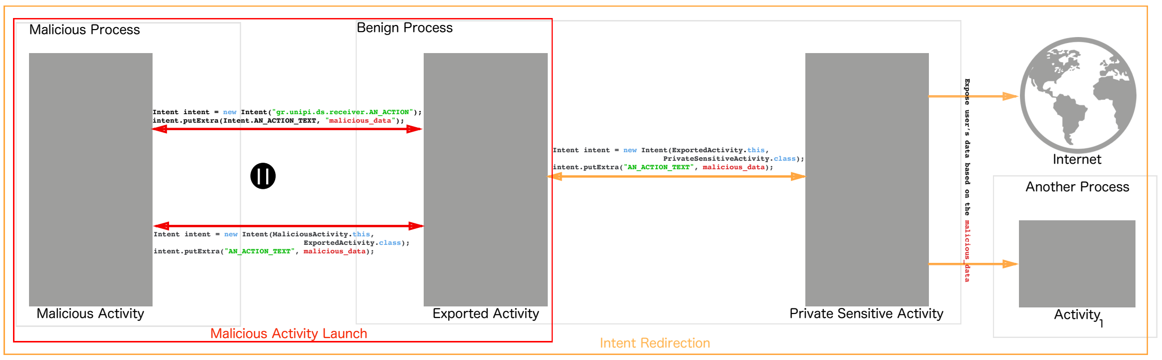
Solution	**Protection Type**		**Limitations**	**Layer**	**Availability**
	IPC Encryption	**Intent Threats** [88, 31]			
Wang *et al.* [88]	No	Yes	Static Policies	Application Development / Framework (Intents)	Yes
Yagemann and Du [95]	No	Yes	Not purely build-in solution	Framework (Intents)	Yes
Kaladharan *et al.* [50]	Yes	No	Only service to application transactions / Not platform generated keys	Kernel (Binder)	No
IntentAuth [68]	Yes	Yes	Limited to implicit Intents (application to application encryption)	Framework (Intents) / Kernel (SELinux)	Yes

Table 6.1 Android Inter-Process Related Threat Prevention Mechanisms Comparison.

6.1.1 Malicious Activity Launch and Intent Hijacking

According to Chin *et al.* [31] a benign application that exports components (*i.e.*, activities) to other applications may be manipulated by a malicious application that performs malicious activity launch attacks. In this case, a malicious application may corrupt the targeted application's storage by sending rogue data or by forcing the execution of specific components which return private or other sensitive data; an action based on the fact that the application receiving intents may not be able to verify the applications using its exported components. This type of attacks is primarily linked with design errors [49] by the application developers, who define activities with public access (public component) that perform sensitive operations over received data or return sensitive information upon their invocation without any restriction, ignoring the recommendations[1] and the best practices that suggest that any activity that performs sensitive operations should be limited to invocations by activities inside the scope of the application they belong. Additionally, the platform allows activities from different applications to securely interact if they are protected through signature-level permissions

[1] https://cwe.mitre.org/data/definitions/926.html



```
Intent intent = new Intent("gr.unipi.ds.receiver.AN_ACTION");
intent.putExtra(Intent.AN_ACTION_TEXT, "malicious_data");
```

```
Intent intent = new Intent(MaliciousActivity.this,
                           ExportedActivity.class);
intent.putExtra("AN_ACTION_TEXT", malicious_data);
```

```
Intent intent = new Intent(ExportedActivity.this,
                           PrivateSensitiveActivity.class);
intent.putExtra("AN_ACTION_TEXT", malicious_data);
```

Figure 6.2 Intent Related Threats in Android Platform.

(activities that belong to different applications signed by the same certificate). Figure 6.2 depicts an example of such a case (refer to the inner part) in which a malicious application invokes directly a benign's exported activity.

At this point it should be stressed that a malicious application may launch another type of attack, called intent hijacking attack, by exporting an activity with the same intent-filter as the benign targeted application, thus trying to trick the user to select it for handling a specific intent request. This is because, whenever an application exports activities with the same intent-filter as other applications, the user is prompted to choose the application he prefers for managing the request. Moreover, whenever a malicious application exports an activity with the same intent-filter as a legitimate one which, however, is not installed on the device, the malicious application will handle the intent message by default, without the user being notified.

6.1.2 Intent Redirection

A variation of the malicious activity launch attack, described above, arises whenever a benign application that exports components, receives intents from untrusted or unknown sources and forwards them to other sensitive components. The exploitation of this vulnerability is known as intent redirection attack [85, 37, 88] or equivalently next-intent vulnerability that generally falls under the confused deputy problem [91]. In such cases, a legitimate application may support a malicious application by forwarding malicious requests to other private components (depicted in the outer part of Figure 6.2).

In other words, a malicious application may launch a legitimate application's private component (via a publicly exported activity that interacts with the private one) with poisoned arguments or arbitrarily launch a legitimate application's component leading to private data leakages. So similarly to the malicious launch attack (see Section 6.1.1), a malevolent application instead of directly invoking another's application sensitive protected activity it can implicitly trigger the execution of a publicly exported application component that communicates and exchange data with the sensitive one, and force the intermediate component to perform a malicious act in it's behalf.

6.2 Enhancement Mechanisms for Android Inter-Process Communication

Chin *et al.*[31] are the first work focusing on flaws against Android Inter-Application communication with emphasis on intents manipulation, demonstrating activity and service hijacking attacks against legitimate applications. In the same direction, Wang *et al.* [88] and Tang *et al.* [85] disclose another class of vulnerability called intent redirection that expose legitimate application's components to data originated from untrusted sources that might cause data leakages. To protect applications and services against these threats various solutions have been proposed in literature varying from (a) static and dynamic analysis detection to (b) run time protection mechanisms.

6.2.1 Static & Dynamic Analysis Detection

CHEX [63] introduces a static analysis tool, the first of its kind, that automatically tests Android applications for component hijacking vulnerabilities. Indeed, CHEX analyzes Android applications and detects possible flaws by conducting low-overhead reachability tests based on customized system dependency graphs. In the same direction, DroidChecker [30] relies on static analysis, however, it builds upon inter-procedural control flow graph and taint analysis.

Complementing such approaches, [70] proposes an Inter-Application Data Flow (IADF) analyzer leveraging on reverse engineering to extract and correlate the under examination application's intents, activities, and manifest features to detect potential leakages that might be caused due to Inter-application communication flaws. Similarly, El–Zawawy *et al.* [37] introduce another static analysis tool for detecting intent redirection vulnerabilities by investigating application's execution sequences.

Instead of relying upon static analysis, IntentFuzzer [96] deploys a dynamic intent fuzzing mechanism, requiring system-level modifications, in order to discover vulnerable applications that can be forced to act on behalf of a malicious application. In a slightly different approach LetterBomb [43] combines static analysis and symbolic execution to automatically generate exploits against Android applications and thus disclose vulnerable Inter-Component

Communication (ICC) implementations, whereas Tang *et al.* [85] uses static and dynamic execution approaches to achieve the same goal.

6.2.2 Runtime Protection

IPC Inspection [42] protects applications against permission re-delegation attacks by introducing a system level runtime mechanism that reduces the privileges of the recipient application to the intersection of the recipient's and requester's permissions. The permission re-delegation attacks concern cases where an application with granted permissions perform privileged tasks on behalf of another application without permission.

Wang *et al.* [88] propose an open-source system-level mechanism, embedded in Android's Intent message mechanism, namely Morbs[2], that can restrict applications' communication according to white-list policies introduced during applications' development. On the contrary, IEM [95] introduced an adaptable policy enforcement mechanism through an intent firewall service that was implemented in the Android framework.

In another scheme, Kaladharan *et al.* [50] propose a system-level encryption mechanism that protects Android's IPC against malicious interceptions. The proposed mechanism builds upon Android's binder inter–process communication mechanism and operates at kernel level, providing confidentiality services to Android applications. However, such an approach does not protect applications against hijacking attacks nor Intent redirection and malicious activity launch attacks.

Lyvas *et al.* [68] introduced a system-level mechanism that provides (a) intents source and/or destination authentication, and (b) confidentiality services for secure IPC between components. Moreover, the proposed solution named *IntentAuth* allows users' to define their policies, regarding which components are allowed to communicated with, in order to gain the control of their data.

Android OS does not support the use of policies for controlling the interactions of an application with others *i.e.*, who is allowed to communicate with whom through implicit intents. In this context, the proposed approach supports this functionality, allowing users to define policies according to their needs.

To do so, a user interface was implemented (as Figure 6.3 depicts) through which a user can specify which application can receive intents from others, by extending Android's framework settings application. The apps that are allowed to communicate can be changed at any time by the user, and thus they are considered dynamic policies. At this point it should be stressed that *IntentAuth* and Android built-in settings application share the same UID as

[2]https://github.com/mobile-security/Morbs

parts of the Android system service ($UID = 1000$). Therefore, *IntentAuth* can obtain user's preferences via settings application at runtime.

By default, *IntentAuth* policies allow any user-installed application to receive implicit intents containing data only from system/vendor applications or themselves while denying any interaction with implicit intents from other user-installed applications, following the least privilege principle. Via settings application, the end users are responsible for enabling or disabling an application's ability to receive intents from another application installed by the user.

The users' policies enable users to control the access to implicit intents requests, however, if this mechanism is not supported by secure communication between the components it can be easily circumvent. In order to ensure secure communication between the components, during the very first execution of a user-space application (invocation of `startActivity(...)` method belonging to `Activity` class), the *IntentAuth* service generates a public/private key pair ($Keychain_n(PU_n, PR_n)$) and a symmetric key ($SecretKey_n$) that will be used for the authentication of components and for data encryption/decryption purposes respectively.

Note that the service can identify whether the appropriate keys have been deployed by checking the existence of them in Android's Keystore, where they are securely stored. Moreover, due to SELinux policy enforcements in the Android Keystore, each application can generate and retrieve keys based on its UID only, and consequently an application cannot retrieve keys from another application if it does not share the same UID or a dedicated policy that allows such an action.

In this context, whenever a user-space application attempts to start an activity that requires the communication with other component(s), through an implicit intent, the *IntentAuth* service, which acts on behalf of the application, retrieves the keys of the corresponding (source) application from the keystore and uses them to sign (with the private key (PR_{source})) and encrypt intent's data (with the symmetric key ($SecretKey_{source}$)), thus enabling the verification (authentication) of the source activity and the protection of the data confidentiality. In fact, intent's initial plaintext is replaced by $SIGN_{PR_{source}(data)}, (ENC_{SecretKey_{source}}(data))$. At that point, the source application is still unaware where the intent is going to be delivered because of its implicit nature. This is the responsibility of the activity task manager which specifies the appropriate activity (destination) for delivering the intent.

As soon as the activity task manager selects the destination activity, the *IntentAuth* consults the user defined policies to check if the communication between the source and the destination applications is allowed or not. If the communication is not allowed the execution is terminated. Otherwise *IntentAuth* validates the origin of the source application, by verifying the signature embedded to the intent's data, and proceeds with the decryption of the intent data, using the source application's symmetric key.

Afterwards, the *IntentAuth* mechanism prior the delivery of the intent to the destination component, it signs and encrypts it (*i.e.*, $SIGN_{PR_{dest}}(data), (ENC_{SecretKey_{dest}}(data))$), in a way similar to the one followed by the source application. Then, the intent is forwarded to the target activity which, upon receipt, verifies the authenticity of the intent. If the verification is successful, it decrypts the intent data $DEC_{SecretKey_{dest}}(ENC_{SecretKey_{dest}}(data)))$. Figure 6.4 overviews this procedure. In this way, *IntentAuth* can achieve a secure delivery from one application to another considering the operating system as a trusted entity.

Moreover, *IntentAuth* was evaluated in terms of the execution overhead imposed to users' applications. *IntentAuth* was deployed on an emulator configured with Android framework version 11 using a proof of concept testbed with applications that communicate over IPC through implicit intents. The results indicate that the applications' execution time increases by 190 ms, at the most, when *IntentAuth* is enabled. The execution overhead mentioned above from the enforcement of *IntentAuth* regards an application to application transmitted intent that contained a 26-byte string data. Of course, the introduced overhead is also affected by the size of data that should be encrypted for secure communication between the processes.

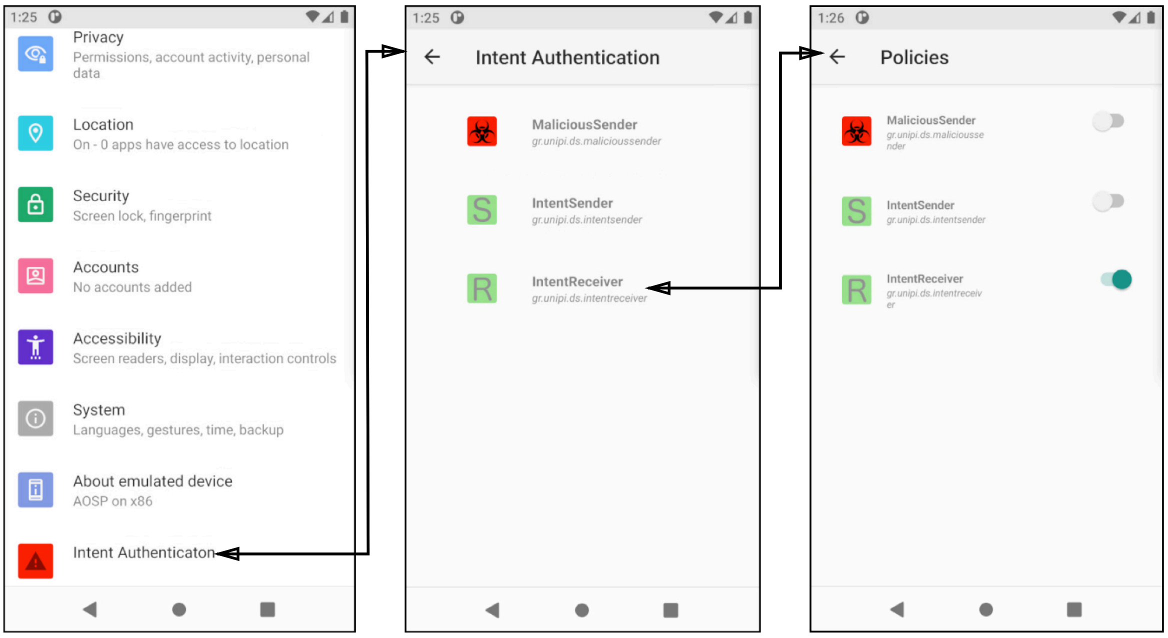
Overall, *IntentAuth* effectively safeguards the integrity, confidentiality, and authenticity of the Inter-Process communication in Android and is implemented as a system service that third party applications can employ for achieving secure communication with other Android components.

IntentAuth key management operations are deployed on top of Android's keystore system that by design is considered as a TEE. Moreover, according to the provisions of the SELinux policy of the Android keystore, applications are not allowed to access keys belonging to other applications; meaning that applications can manage keys belonging only to the same UID.

However, due to its intrinsic characteristics, it requires access to application keys. Thus, an additional SELinux policy was deployed for Android Keychain that allows system service ($UID = 1000$) to gain access to user-space applications ($10000 \leq UID \leq 19999$)[3] keys. Therefore, *IntentAuth* was developed as an operating system service, and hence it can be considered a trusted component similar to other OS components.

Indeed, with the support of such a mechanism, *IntentAuth* gains access only to the key chains and symmetric keys of the involved user-space applications without exposing them to other user-space applications. As such, *IntentAuth* safeguards the integrity, confidentiality, and authenticity of the keys required for the secure communication of the intents.

[3]https://android.googlesource.com/platform/system/core/+/refs/tags/android-11.0.0_r3/libcutils/include/private/android_filesystem_config.h


1:25

Privacy
Permissions, account activity, personal data

Location
On - 0 apps have access to location

Security
Screen lock, fingerprint

Accounts
No accounts added

Accessibility
Screen readers, display, interaction controls

System
Languages, gestures, time, backup

About emulated device
AOSP on x86

Intent Authenticaton

1:25

Intent Authentication

MaliciousSender
gr.unipi.ds.malicioussender

IntentSender
gr.unipi.ds.intentsender

IntentReceiver
gr.unipi.ds.intentreceiver

1:26

Policies

MaliciousSender
gr.unipi.ds.malicioussender

IntentSender
gr.unipi.ds.intentsender

IntentReceiver
gr.unipi.ds.intentreceiver

Figure 6.3 *IntentAuth* User-Defined Policies via Android (Version 11) Settings Application.

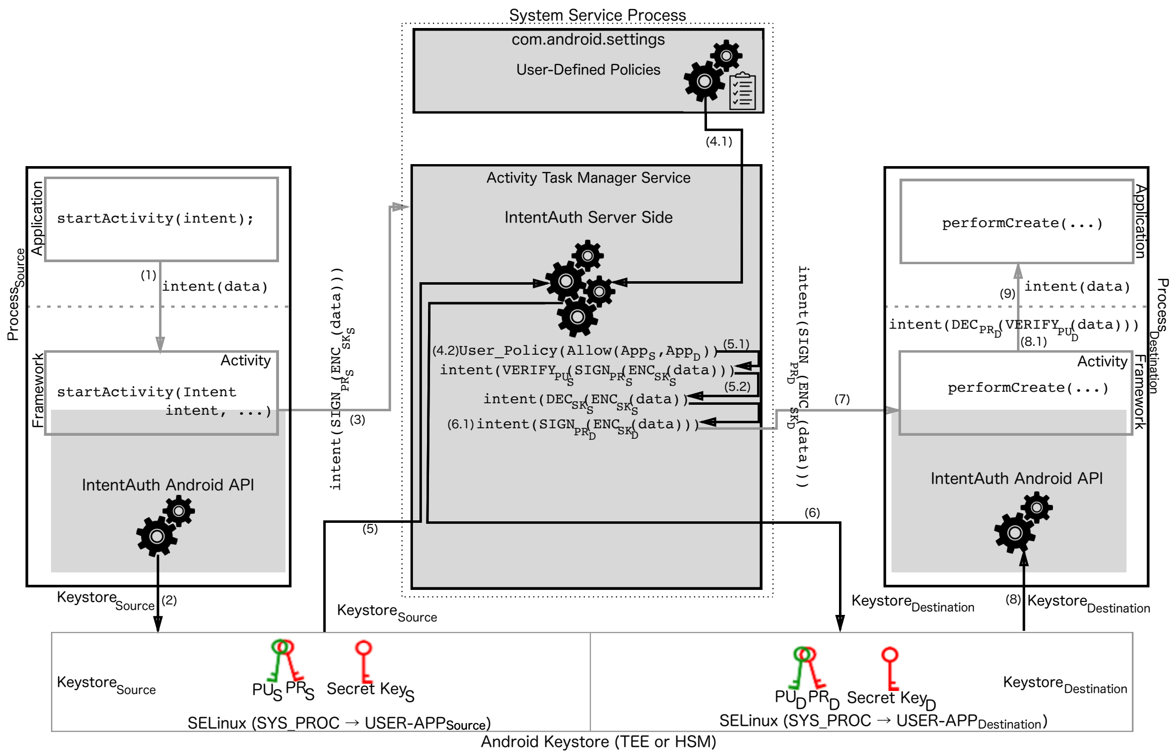


Figure 6.4 *IntentAuth* High-Level Overview.

Chapter 7

Conclusions

Android is one of the dominant mobile operating systems with millions of applications available for its plethora of users. Several system and application-level mechanisms protect the security of the devices and the privacy of the users. However, there are several threats against user's privacy and security due to the platform's peculiarities. This dissertation copes with the permission map compilation and the threats of task, activity, intent hijacking, malicious activity launch, and the intent redirection. As such threats are eminent security flaws requiring high attention, several mechanisms have been introduced to identify deficiencies, discover vulnerabilities, and prevent exploitation in this thesis.

For the Android permission map generation, the *Dypermin* tool was introduced based on a method capable of compiling the permission map for any given Android version without generating any false positives. In addition, *Dypermin* is transparent to the underlying operating system, meaning that there is no need for any operating system modifications.

Additionally, *Anactijax* was deployed to identify vulnerable applications and configurations regarding task and activity hijacking threats. Moreover, the system level prevention mechanism *TaskAuth* was introduced that controls activities interactions based on predefined dynamic policies. *TaskAuth* demands system-level modifications, but it is transparent both to end-users and developers.

Finally, regarding the threats of intent hijacking, malicious activity launch, and intent redirection, a system-based security service named *IntentAuth* was introduced to enable encrypted IPC between applications and provide users the capacity to control applications' IPC through adaptable policies. Although *IntentAuth* requires system-level modifications, it is completely transparent to applications and users and does not impose any significant overhead on users' experience navigation.

We are currently focusing on prioritizing the Android framework threats, aiming to extend our work to protect all forms of the Inter-Process Communication scheme of the Android platform with the redesign of the presented in this thesis prevention mechanism of *IntentAuth* without the need for underlying operating system modification able to provide a combined trust model with dynamic policies given by both developers and end-users.

Bibliography

[1] Android Open Source Project. Application Fundamentals. Technical report, 2019. URL https://developer.android.com/guide/components/fundamentals. Cited March 22nd 2020.

[2] Android Open Source Project. Intents and Intent Filters. Technical report, 2019. URL https://developer.android.com/guide/components/intents-filters. Cited March 22nd 2020.

[3] Android Open Source Project. Processes and Threads Overview. Technical report, 2019. URL https://developer.android.com/guide/components/processes-and-threads. Cited March 25th 2020.

[4] Android Open Source Project. Understand Tasks and Back Stack. Technical report, 2019. URL https://developer.android.com/guide/components/activities/tasks-and-back-stack. Cited March 22nd 2020.

[5] Android Open Source Project. Implementing SELinux. Technical report, 2020. URL https://source.android.com/security/selinux/implement. Cited September 23rd 2020.

[6] Android Open Source Project. Security Enhancements in Android 1.5 through 4.1. Technical report, 2020. URL https://source.android.com/security/enhancements/enhancements41. Cited September 23rd 2020.

[7] Android Open Source Project. Security and Privacy Enhancements in Android 10. Technical report, 2020. URL https://source.android.com/security/enhancements/enhancements10. Cited September 23rd 2020.

[8] Android Open Source Project. Security Enhancements in Android 4.2. Technical report, 2020. URL https://source.android.com/security/enhancements/enhancements42. Cited September 23rd 2020.

[9] Android Open Source Project. Security Enhancements in Android 4.3. Technical report, 2020. URL https://source.android.com/security/enhancements/enhancements43. Cited September 23rd 2020.

[10] Android Open Source Project. Security Enhancements in Android 4.4. Technical report, 2020. URL https://source.android.com/security/enhancements/enhancements44. Cited September 23rd 2020.

[11] Android Open Source Project. Security Enhancements in Android 5.0. Technical report, 2020. URL https://source.android.com/security/enhancements/enhancements50. Cited September 23rd 2020.

[12] Android Open Source Project. Security Enhancements in Android 6.0. Technical report, 2020. URL https://source.android.com/security/enhancements/enhancements60. Cited September 23rd 2020.

[13] Android Open Source Project. Security Enhancements in Android 7.0. Technical report, 2020. URL https://source.android.com/security/enhancements/enhancements70. Cited September 23rd 2020.

[14] Android Open Source Project. Security Enhancements in Android 8.0. Technical report, 2020. URL https://source.android.com/security/enhancements/enhancements80. Cited September 23rd 2020.

[15] Android Open Source Project. Security Enhancements in Android 9.0. Technical report, 2020. URL https://source.android.com/security/enhancements/enhancements9. Cited September 23rd 2020.

[16] Android Open Source Project. Android Architecture. Technical report, 2020. URL https://source.android.com/devices/architecture. Cited March 24th 2020.

[17] Android Open Source Project. Gatekeeper. Technical report, 2020. URL https://source.android.com/security/authentication/gatekeeper. Cited October 10th 2020.

[18] Android Open Source Project. Android Gradle plugin release notes. Technical report, 2020. URL https://developer.android.com/studio/releases/gradle-plugin. Cited October 29th 2020.

[19] Android Open Source Project. Android Keystore System. Technical report, 2020. URL https://developer.android.com/training/articles/keystore. Cited October 30th 2020.

[20] Android Open Source Project. Permissions on Android. Technical report, 2020. URL https://developer.android.com/guide/topics/permissions/overview. Cited September 23rd 2020.

[21] Android Open Source Project. Application Signing. Technical report, 2020. URL https://source.android.com/security/apksigning. Cited Octomber 20th 2020.

[22] Android Open Source Project. Trusty TEE. Technical report, 2020. URL https://source.android.com/security/trusty. Cited September 23rd 2020.

[23] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, and Konrad Rieck. Drebin: Effective and explainable detection of android malware in your pocket. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2013*, 2014.

[24] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. Pscout: analyzing the android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, pages 217–228, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1651-4.

[25] Michael Backes, Sven Bugiel, Erik Derr, Patrick McDaniel, Damien Octeau, and Sebastian Weisgerber. On demystifying the android application framework: Re-visiting android permission specification analysis. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 1101–1118, Austin, TX, 2016. USENIX Association. ISBN 978-1-931971-32-4.

[26] Alexandre Bartel, Jacques Klein, Martin Monperrus, and Yves Le Traon. Static analysis for extracting permission checks of a large scale framework: The challenges and solutions for analyzing android. *IEEE Transactions on Software Engineering (TSE)*, 2014.

[27] Parnika Bhat and Kamlesh Dutta. A survey on various threats and current state of security in android platform. *ACM Computing Surveys (CSUR)*, 52(1):1–35, 2019.

[28] Andrew D. Birrell and Bruce Jay Nelson. Implementing Remote Procedure Calls. *ACM Trans. Comput. Syst.*, 2(1):39–59, February 1984. ISSN 0734-2071. doi: 10.1145/2080.357392. URL https://doi.org/10.1145/2080.357392.

[29] Denis Bogdanas. Dperm: Assisting the migration of android apps to runtime permissions. *CoRR*, abs/1706.05042, 2017. URL http://arxiv.org/abs/1706.05042.

[30] Patrick PF Chan, Lucas CK Hui, and Siu-Ming Yiu. Droidchecker: analyzing android applications for capability leak. In *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*, pages 125–136, 2012.

[31] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing inter-application communication in android. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, pages 239–252, 2011.

[32] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. Privilege escalation attacks on android. In *international conference on Information security*, pages 346–360. Springer, 2010.

[33] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *ECOOP '95 Proceedings of the 9th European Conference on Object-Oriented Programming*, ECOOP '95, pages 77–101, London, UK, 1995. Springer-Verlag. ISBN 3-540-60160-0.

[34] XDA Developers. Xposed framework hub, 2020. URL https://www.xda-developers.com/xposed-framework-hub/.

[35] Vasiliki Diamantopoulou, Christos Kalloniatis, Christos Lyvas, Konstantinos Maliatsos, Matthieu Gay, Athanasios Kanatas, and Costas Lambrinoudakis. Aligning the concepts of risk, security and privacy towards the design of secure intelligent transport systems. In *Computer Security*, pages 170–184. Springer, 2020. doi: https://doi.org/10.1007/978-3-030-64330-0_11. URL https://link.springer.com/chapter/10.1007/978-3-030-64330-0_11.

[36] Yu Ding, Zhuo Peng, Yuanyuan Zhou, and Chao Zhang. Android low entropy demystified. In *2014 IEEE International Conference on Communications (ICC)*, pages 659–664. IEEE, 2014.

[37] Mohamed A El-Zawawy, Eleonora Losiouk, and Mauro Conti. Do not let next-intent vulnerability be your next nightmare: type system-based approach to detect it in android apps. *International Journal of Information Security*, pages 1–20, 2020.

[38] William Enck, Machigar Ongtang, and Patrick McDaniel. Understanding android security. *IEEE Security and Privacy*, 7(1):50–57, January 2009. ISSN 1540-7993. doi: 10.1109/MSP.2009.26. URL `http://dx.doi.org/10.1109/MSP.2009.26`.

[39] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid: An information flow tracking system for real-time privacy monitoring on smartphones. *Commun. ACM*, 57(3):99–106, March 2014. ISSN 0001-0782. doi: 10.1145/2494522. URL `https://doi.org/10.1145/2494522`.

[40] P. Faruki, A. Bharmal, V. Laxmi, V. Ganmoor, M. S. Gaur, M. Conti, and M. Rajarajan. Android security: A survey of issues, malware penetration, and defenses. *IEEE Communications Surveys Tutorials*, 17(2):998–1022, 2015.

[41] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 627–638, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0948-6.

[42] Adrienne Porter Felt, Helen J Wang, Alexander Moshchuk, Steve Hanna, and Erika Chin. Permission re-delegation: Attacks and defenses. In *USENIX security symposium*, volume 30, page 88, 2011.

[43] Joshua Garcia, Mahmoud Hammad, Negar Ghorbani, and Sam Malek. Automatic generation of inter-component communication exploits for android applications. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 661–671, 2017.

[44] Dimitris Geneiatakis, Igor Nai Fovino, Ioannis Kounelis, and Paquale Stirparo. A permission verification approach for android mobile applications. *Computers & Security*, 49:192–205, 2015.

[45] Hector Marco Gisbert and Ismael Ripoll. On the effectiveness of nx, ssp, renewssp, and aslr against stack buffer overflows. In *2014 IEEE 13th International Symposium on Network Computing and Applications*, pages 145–152. IEEE, 2014.

[46] Google LLC. Android Enterprise Security White Paper. Technical report, 2020. URL `https://static.googleusercontent.com/media/www.android.com/en//static/2016/pdfs/enterprise/Android_Enterprise_Security_White_Paper_2019.pdf`. Cited March 22nd 2020.

[47] Norm Hardy. The confused deputy: (or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review*, 22(4):36–38, 1988.

[48] Sungjae Hwang, Sungho Lee, and Sukyoung Ryu. All about activity injection: Threats, semantics, detection, and defense. *Software: Practice and Experience*, 50(7):1061–1086, 2020. doi: 10.1002/spe.2792. URL `https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2792`.

[49] Ajay Kumar Jha, Sunghee Lee, and Woo Jin Lee. Developer mistakes in writing android manifests: An empirical study of configuration errors. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 25–36, 2017. doi: 10.1109/MSR.2017.41.

[50] Yadu Kaladharan, Prabhaker Mateti, and KP Jevitha. An encryption technique to thwart android binder exploits. In *Intelligent Systems Technologies and Applications*, pages 13–21. Springer, 2016.

[51] Christos Kalloniatis, Vasiliki Diamantopoulou, Konstantinos Kotis, Christos Lyvas, Konstantinos Maliatsos, Matthieu Gay, Athanasios G Kanatas, and Costas Lambrinoudakis. Towards the design of an assurance framework for increasing security and privacy in connected vehicles. *International Journal of Internet of Things and Cyber-Assurance*, 1(3-4):244–266, 2020. ISSN 2059-7967. doi: https://doi.org/10.1504/IJITCA.2020.112574. URL https://www.inderscienceonline.com/doi/abs/10.1504/IJITCA.2020.112574.

[52] Anatoli Kalysch, Mark Deutel, and Tilo Müller. Template-based android inter process communication fuzzing. In *Proceedings of the 15th International Conference on Availability, Reliability and Security*, pages 1–6, 2020.

[53] Foutse Khomh, Hao Yuan, and Ying Zou. Adapting Linux for mobile platforms: An empirical study of Android. In *2012 28th IEEE international conference on software maintenance (ICSM)*, pages 629–632. IEEE, 2012.

[54] Vasileios Kouliaridis, Georgios Kambourakis, Dimitris Geneiatakis, and Nektaria Potha. Two anatomists are better than one—dual-level android malware detection. *Symmetry*, 12(7), 2020. ISSN 2073-8994. doi: 10.3390/sym12071128. URL https://www.mdpi.com/2073-8994/12/7/1128.

[55] Sungho Lee, Sungjae Hwang, and Sukyoung Ryu. All about activity injection: threats, semantics, and detection. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 252–262. IEEE, 2017.

[56] Ondřej Lhoták and Laurie Hendren. Scaling java points-to analysis using spark. In *Proceedings of the 12th international conference on Compiler construction*, CC'03, pages 153–169, New York, NY, USA, 2003. ACM. ISBN 3-540-00904-3.

[57] Jin Li, Lichao Sun, Qiben Yan, Zhiqiang Li, Witawas Srisa-An, and Heng Ye. Significant permission identification for machine-learning-based android malware detection. *IEEE Transactions on Industrial Informatics*, 14(7):3216–3225, 2018.

[58] Li Li, Tegawendé F Bissyandé, Yves Le Traon, and Jacques Klein. Accessing inaccessible android apis: An empirical study. In *The 32nd International Conference on Software Maintenance and Evolution (ICSME 2016)*, 2016.

[59] Li Li, Tegawendé F Bissyandé, Damien Octeau, and Jacques Klein. Droidra: taming reflection to support whole-program analysis of android apps. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 318–329, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4390-9.

[60] Steffen Liebergeld and Matthias Lange. Android security, pitfalls and lessons learned. In *Information Sciences and Systems 2013*, pages 409–417. Springer, 2013.

[61] Jie Liu, Diyu Wu, and Jingling Xue. Tdroid: exposing app switching attacks in android with control flow specialization. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 236–247, 2018.

[62] Kangjie Lu, Chengyu Song, Byoungyoung Lee, Simon P. Chung, Taesoo Kim, and Wenke Lee. Aslr-guard: Stopping address space leakage for code reuse attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 280–291, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450338325. doi: 10.1145/2810103.2813694. URL `https://doi.org/10.1145/2810103.2813694`.

[63] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. Chex: statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 229–240, 2012.

[64] Christos Lyvas, Nikolaos Pitropakis, and Costas Lambrinoudakis. The far side of mobile application integrated development environments. In *International Conference on Trust and Privacy in Digital Business*, pages 111–122. Springer, 2016. doi: https://doi.org/10.1007/978-3-319-44341-6_8. URL `https://link.springer.com/chapter/10.1007/978-3-319-44341-6_8`.

[65] Christos Lyvas, Costas Lambrinoudakis, and Dimitris Geneiatakis. Dypermin: Dynamic permission mining framework for android platform. *Computers & Security*, 77: 472–487, 2018. ISSN 0167-4048. doi: https://doi.org/10.1016/j.cose.2018.05.007. URL `https://www.sciencedirect.com/science/article/pii/S0167404818304954`.

[66] Christos Lyvas, Costas Lambrinoudakis, and Dimitris Geneiatakis. On android's activity hijacking prevention. *Computers & Security*, 111:102468, 2021. ISSN 0167-4048. doi: https://doi.org/10.1016/j.cose.2021.102468. URL `https://www.sciencedirect.com/science/article/pii/S0167404821002923`.

[67] Christos Lyvas, Christoforos Ntantogian, and Christos Xenakis. [m]allotropism: a metamorphic engine for malicious software variation development. *International Journal of Information Security*, pages 1–18, 2021. doi: https://doi.org/10.1007/s10207-021-00541-y. URL `https://link.springer.com/article/10.1007/s10207-021-00541-y`.

[68] Christos Lyvas, Costas Lambrinoudakis, and Dimitris Geneiatakis. IntentAuth: Securing Android's Intent Based Inter-Process Communication. *International Journal of Information Security*, 2021 (Under Minor Revision).

[69] Konstantinos Maliatsos, Christos Lyvas, Panagiotis Pantazopoulos, Costas Lambrinoudakis, Athanasios Kanatas, Matthieu Gay, and Angelos Amditis. Standardizing security evaluation criteria for connected vehicles: A modular protection profile. In *2019 IEEE Conference on Standards for Communications and Networking (CSCN)*, pages 1–7. IEEE, 2019. doi: https://doi.org/10.1109/CSCN.2019.8931344. URL `https://ieeexplore.ieee.org/document/8931344`.

[70] Hafiz Muhammad Arslan Maqsood, Kashif Naseer Qureshi, Faisal Bashir, and Najam Ul Islam. Privacy leakage through exploitation of vulnerable inter-app communication on android. In *2019 13th International Conference on Open Source Systems and Technologies (ICOSST)*, pages 1–6. IEEE, 2019.

[71] René Mayrhofer, Jeffrey Vander Stoep, Chad Brubaker, and Nick Kralevich. The Android Platform Security Model. *ACM Trans. Priv. Secur.*, 24(3), April 2021. ISSN 2471-2566. doi: 10.1145/3448609. URL https://doi.org/10.1145/3448609.

[72] Niall McLaughlin, Jesus Martinez del Rincon, BooJoong Kang, Suleiman Yerima, Paul Miller, Sakir Sezer, Yeganeh Safaei, Erik Trickel, Ziming Zhao, Adam Doupé, and Gail Joon Ahn. Deep android malware detection. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, CODASPY '17, page 301–308, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450345231. doi: 10.1145/3029806.3029823. URL https://doi.org/10.1145/3029806.3029823.

[73] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for java. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '02, pages 1–11, New York, NY, USA, 2002. ACM. ISBN 1-58113-562-9.

[74] Jon Oberheide and Charlie Miller. Dissecting the android bouncer. *SummerCon2012, New York*, 95:110, 2012.

[75] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. Effective inter-component communication mapping in android: An essential step towards holistic security analysis. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 543–558, 2013.

[76] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *ICSE'07, Proceedings of the 29th International Conference on Software Engineering*, pages 75–84, Minneapolis, MN, USA, May 23–25, 2007.

[77] Naser Peiravian and Xingquan Zhu. Machine learning for android malware detection using permission and api calls. In *2013 IEEE 25th international conference on tools with artificial intelligence*, pages 300–305. IEEE, 2013.

[78] Nikolaos Pitropakis, Christos Lyvas, and Costas Lambrinoudakis. The greater the power, the more dangerous the abuse: facing malicious insiders in the cloud. In *International Conference on Cloud Computing, GRIDs, and Virtualization, Track Security and Privacy in Cloud Computing (SEPRICC 2017)*, pages 156–161. IARIA Xpert Publishing Services, 2017.

[79] Chuangang Ren, Yulong Zhang, Hui Xue, Tao Wei, and Peng Liu. Towards discovering and understanding task hijacking in android. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 945–959, 2015.

[80] Chuangang Ren, Peng Liu, and Sencun Zhu. Windowguard: Systematic protection of gui security in android. In *2017 Network and Distributed System Security (NDSS) Symposium*, 2017.

[81] Yunlong Ren, Yue Li, Fangfang Yuan, and Fangjiao Zhang. Hijacking activity technology analysis and research in android system. In *International Conference on Trustworthy Computing and Services*, pages 46–53. Springer, 2013.

[82] Stephen Smalley and Robert Craig. Security Enhanced (SE) Android: Bringing Flexible MAC to Android. In *NDSS*, volume 310, pages 20–38, 2013.

[83] Patrícia Gomes Soares. On remote procedure call. In *Proceedings of the 1992 Conference of the Centre for Advanced Studies on Collaborative Research - Volume 2*, CASCON '92, page 215–267. IBM Press, 1992.

[84] Geran Tam and Aaron Hunter. Machine learning to identify android malware. In *2018 9th IEEE Annual Ubiquitous Computing, Electronics & Mobile Communication Conference (UEMCON)*, pages 1–5. IEEE, 2018.

[85] Junjie Tang, Xingmin Cui, Ziming Zhao, Shanqing Guo, Xinshun Xu, Chengyu Hu, Tao Ban, and Bing Mao. Nivanalyzer: a tool for automatically detecting and verifying next-intent vulnerabilities in android apps. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 492–499. IEEE, 2017.

[86] Ximing Tang, Tao Song, Kun Wang, and Alei Liang. Fine-Grained Access Control on Android Through Behavior Monitoring. In *Advances in Computer Communication and Computational Sciences*, pages 525–532. Springer, 2019.

[87] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *CASCON 99, Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*. IBM Press, 1999.

[88] Rui Wang, Luyi Xing, XiaoFeng Wang, and Shuo Chen. Unauthorized origin crossing on mobile platforms: Threats and mitigation. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 635–646, 2013.

[89] Zhaoguo Wang, Chenglong Li, Yi Guan, Yibo Xue, and Yingfei Dong. Activityhijacker: Hijacking the android activity component for sensitive data. In *2016 25th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–9. IEEE, 2016.

[90] Xuetao Wei, Lorenzo Gomez, Iulian Neamtiu, and Michalis Faloutsos. Permission evolution in the android ecosystem. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 31–40, 2012.

[91] Jianliang Wu, Tingting Cui, Tao Ban, Shanqing Guo, and Lizhen Cui. Paddyfrog: systematically detecting confused deputy vulnerability in android applications. *Security and Communication Networks*, 8(13):2338–2349, 2015.

[92] Min-Hao Wu, Fu-Hau Hsu, Ting-Cheng Chang, and Liang Hsuan Lin. Preventing activity hijacking attacks in android app. In *2019 IEEE Eurasia Conference on Biomedical Engineering, Healthcare and Sustainability (ECBIOS)*, pages 170–173. IEEE, 2019.

[93] Yinhao Xiao, Guangdong Bai, Jian Mao, Zhenkai Liang, and Wei Cheng. Privilege leakage and information stealing through the android task mechanism. In *2017 IEEE Symposium on Privacy-Aware Computing (PAC)*, pages 152–163. IEEE, 2017.

[94] Luyi Xing, Xiaorui Pan, Rui Wang, Kan Yuan, and XiaoFeng Wang. Upgrading your android, elevating my malware: Privilege escalation through mobile os updating. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 393–408, Washington, DC, USA, 2014. IEEE Computer Society. ISBN 978-1-4799-4686-0. doi: 10.1109/SP.2014.32. URL http://dx.doi.org/10.1109/SP.2014.32.

[95] Carter Yagemann and Wenliang Du. Intentio ex machina: Android intent access control via an extensible application hook. In *European Symposium on Research in Computer Security*, pages 383–400. Springer, 2016.

[96] Kun Yang, Jianwei Zhuge, Yongke Wang, Lujue Zhou, and Haixin Duan. Intentfuzzer: detecting capability leaks of android applications. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pages 531–536, 2014.

[97] Ayman Youssef and Ahmed F. Shosha. Quantitave dynamic taint analysis of privacy leakage in android arabic apps. In *Proceedings of the 12th International Conference on Availability, Reliability and Security*, ARES '17, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450352574. doi: 10.1145/3098954.3105827. URL https://doi.org/10.1145/3098954.3105827.

[98] Jianing Zhang, Xingtao Zhuang, and Yunfang Chen. Android malware detection combined with static and dynamic analysis. In *Proceedings of the 2019 the 9th International Conference on Communication and Network Security*, ICCNS 2019, page 6–10, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450376624. doi: 10.1145/3371676.3371685. URL https://doi.org/10.1145/3371676.3371685.

[99] Yuan Zhang, Min Yang, Bingquan Xu, Zhemin Yang, Guofei Gu, Peng Ning, X. Sean Wang, and Binyu Zang. Vetting undesirable behaviors in android apps with permission use analysis. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer &#38; Communications Security*, CCS '13, pages 611–622, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2477-9. doi: 10.1145/2508859.2516689.

[100] Yury Zhauniarovich and Olga Gadyatskaya. *Small Changes, Big Changes: An Updated View on the Android Permission System*, pages 346–367. Springer International Publishing, Cham, 2016. ISBN 978-3-319-45719-2. doi: 10.1007/978-3-319-45719-2_16. URL https://doi.org/10.1007/978-3-319-45719-2_16.

Appendix A

Intents Usage Examples

A.1 Implicit Intents

```
...
<activity android:name=".SecondActivity">
      <intent-filter>
         <action android:name="SECOND_ACTIVITY"/>
         <category android:name="android.intent.category.DEFAULT"/>
         <data android:mimeType="text/plain"/>
      </intent-filter>
    ...
</activity>
...
```

Listing A.1 Declaration of Activity Initiated by Implicit Intent.

```
  ...
  Intent intent = new Intent("SECOND_ACTIVITY");
  intent.putExtra(Intent.EXTRA_TEXT, new String("finalText");
  intent.setType("text/plain");
  startActivity(intent);
```

Listing A.2 Invocation of an Activity with an Implicit Intent.

A.2 Explicit Intents

```
...
<activity android:name=".SecondActivity">
  ...
```

```
</activity>
...
```

Listing A.3 Declaration of Activity Initiated by Explicit Intent.

```
  ...
  Intent intent = new Intent(this, SecondActivity.class);
  intent.putExtra("EXTRA_TEXT", new String("finalText"));
  startActivity(intent);
```

Listing A.4 Invocation of an Activity with an Explicit Intent.

Appendix B

Anactijax Generated Use Cases Examples

B.1 Activity Injection Against any Benign Application

```
...
<activity
      android:exported="true"
      android:windowSoftInputMode="stateHidden"
      android:launchMode="singleTask"
      android:taskAffinity="gr.unipi.ds.victim"
      android:name="gr.unipi.ds.hijacker.MainActivity2">
</activity>
...
```

Listing B.1 Example of Manifest File Generated by *Anactijax* for Malicious Activity Injection Against a Default Benign Application.

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

Listing B.2 Example of Manifest File Generated by *Anactijax* for Default Vulnerable Benign Application.

B.2 Activity Injection Against Vulnerable Benign Application's Tasks

```
...
<activity
      android:exported="true"
      android:taskAffinity="gr.unipi.ds.victim.second"
      android:name="gr.unipi.ds.hijacker.MainActivity2">
</activity>
...
```

Listing B.3 Example of Manifest File Generated by *Anactijax* for Malicious Activity Injection Against a Vulnerable Benign Application with Explicitly Defined Tasks.

```
  ...
  Intent intent = new Intent(MainActivity.this, MainActivity2.class);
  intent.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
  startActivity(intent);
```

Listing B.4 Example of Code Generated by *Anactijax* for Malicious Activity Injection Against a Vulnerable Benign Application with Explicitly Defined Tasks.

```
...
<activity
      android:name="gr.unipi.ds.victim.SecondActivity"
      android:taskAffinity="gr.unipi.ds.victim.second">
      ...
</activity>
...
```

Listing B.5 Example of Manifest File Generated by *Anactijax* for Vulnerable Benign Application with Explicitly Defined Tasks.

Index

Curriculum Vitae

Christos Lyvas received his B.Sc. from the Department of Informatics, Athens University of Economics and Business (AUEB), in 2013, and his M.Sc. in Digital Systems Security from the Department of Digital Systems, University of Piraeus, in 2015. In 2015 he served the Hellenic army as a security analyst at the Hellenic Army Information Technology Support Center, Hellenic Army General Staff. Since March 2016, he has been a member of the Systems Security Laboratory (SSL) of the University of Piraeus. He has participated as security analyst in several European research programs, such as "SAFERtec", "BIONIC", "CitySCAPE", and "RE-SAMPLE" within the implementation and innovation program "Horizon 2020" of European Commission. His research focuses on mobile applications and operating system's security, malware analysis, and Internet-of-Things security. He is a member of the IEEE and certified ISO 27001:2013 Lead Auditor by TÜV Hellas (TÜV Nord).

Publications

Journals

Christos Kalloniatis, Vasiliki Diamantopoulou, Konstantinos Kotis, Christos Lyvas, Konstantinos Maliatsos, Matthieu Gay, Athanasios G Kanatas, and Costas Lambrinoudakis. Towards the design of an assurance framework for increasing security and privacy in connected vehicles. *International Journal of Internet of Things and Cyber-Assurance*, 1(3-4):244–266, 2020. ISSN 2059-7967. doi: https://doi.org/10.1504/IJITCA.2020.112574. URL https://www.inderscienceonline.com/doi/abs/10.1504/IJITCA.2020.112574.

Christos Lyvas, Costas Lambrinoudakis, and Dimitris Geneiatakis. Dypermin: Dynamic permission mining framework for android platform. *Computers & Security*, 77:472–487, 2018. ISSN 0167-4048. doi: https://doi.org/10.1016/j.cose.2018.05.007. URL https://www.sciencedirect.com/science/article/pii/S0167404818304954.

Christos Lyvas, Costas Lambrinoudakis, and Dimitris Geneiatakis. On android's activity hijacking prevention. *Computers & Security*, 111:102468, 2021. ISSN 0167-4048. doi: https://doi.org/10.1016/j.cose.2021.102468. URL https://www.sciencedirect.com/science/article/pii/S0167404821002923.

Christos Lyvas, Christoforos Ntantogian, and Christos Xenakis. [m]allotropism: a metamorphic engine for malicious software variation development. *International Journal of Information Security*, pages 1–18, 2021. doi: https://doi.org/10.1007/s10207-021-00541-y. URL https://link.springer.com/article/10.1007/s10207-021-00541-y.

Christos Lyvas, Costas Lambrinoudakis, and Dimitris Geneiatakis. IntentAuth: Securing Android's Intent Based Inter-Process Communication. *International Journal of Information Security*, 2021 (Under Minor Revision).

Conferences

Vasiliki Diamantopoulou, Christos Kalloniatis, Christos Lyvas, Konstantinos Maliatsos, Matthieu Gay, Athanasios Kanatas, and Costas Lambrinoudakis. Aligning the concepts of risk, security and privacy towards the design of secure intelligent transport systems. In *Computer Security*, pages 170–184. Springer, 2020. doi: https://doi.org/10.1007/978-3-030-64330-0_11. URL https://link.springer.com/chapter/10.1007/978-3-030-64330-0_11.

Christos Lyvas, Nikolaos Pitropakis, and Costas Lambrinoudakis. The far side of mobile application integrated development environments. In *International Conference on Trust and Privacy in Digital Business*, pages 111–122. Springer, 2016. doi: https://doi.org/10.1007/978-3-319-44341-6_8. URL https://link.springer.com/chapter/10.1007/978-3-319-44341-6_8.

Konstantinos Maliatsos, Christos Lyvas, Panagiotis Pantazopoulos, Costas Lambrinoudakis, Athanasios Kanatas, Matthieu Gay, and Angelos Amditis. Standardizing security evaluation criteria for connected vehicles: A modular protection profile. In *2019 IEEE Conference on Standards for Communications and Networking (CSCN)*, pages 1–7. IEEE, 2019. doi: https://doi.org/10.1109/CSCN.2019.8931344. URL https://ieeexplore.ieee.org/document/8931344.

Nikolaos Pitropakis, Christos Lyvas, and Costas Lambrinoudakis. The greater the power, the more dangerous the abuse: facing malicious insiders in the cloud. In *International Conference on Cloud Computing, GRIDs, and Virtualization, Track Security and Privacy in Cloud Computing (SEPRICC 2017)*, pages 156–161. IARIA Xpert Publishing Services, 2017.