Πανεπιστήμιο Πειραιώς – Τμήμα Πληροφορικής

Πρόγραμμα Μεταπτυχιακών Σπουδών

«Πληροφορική»

Μεταπτυχιακή Διατριβή

Τίτλος Διατριβής	**Αυτοματοποιημένος έλεγχος ασφαλείας του Android API μέσω fuzzing** **Automated security testing of Android API using fuzzing**
Ονοματεπώνυμο Φοιτητή	**Βασιλική Πουλάκη**
Πατρώνυμο	**Δημήτριος**
Αριθμός Μητρώου	**ΜΠΠΛ/ 14070**
Επιβλέπων	**Κωνσταντίνος Πατσάκης, Επίκουρος Καθηγητής**

Ημερομηνία Παράδοσης **Δεκέμβριος 2018**

Τριμελής Εξεταστική Επιτροπή

Πατσάκης Κωνσταντίνος Επίκουρος Καθηγητής	Αλέπης Ευθύμιος Επίκουρος Καθηγητής	Τασούλας Ιωάννης Επίκουρος Καθηγητής

Περίληψη

Η ανάπτυξη της τεχνολογίας των έξυπνων κινητών τηλεφώνων, καθώς και η αλματώδης εξέλιξη των λειτουργικών συστημάτων τους, δημιουργούν τεράστιες δυνατότητες στους προγραμματιστές για ανάπτυξη εφαρμογών. Ταυτόχρονα όμως τα λειτουργικά αυτά συστήματα εκτίθενται σε κινδύνους έναντι κακόβουλων λογισμικών. Η διεπαφή που παρέχει το λειτουργικό σύστημα **Android** της **Google** στους προγραμματιστές για ανάπτυξη εφαρμογών ανανεώνεται συνεχώς, παρέχοντας ποικίλες δυνατότητες αξιοποίησης των λειτουργικοτήτων της συσκευής. Το γεγονός αυτό σε συνδυασμό με την οικονομική προσιτότητα των κινητών συσκευών Android έχει εκτοξεύσει την αγορά της ανάπτυξης εφαρμογών που υλοποιούνται για το συγκεκριμένο λειτουργικό σύστημα, αλλά ταυτόχρονα έχει ανοίξει διόδους εκμετάλευσης αυτών με κακόβουλο σκοπό. Το σύστημα παροχής αδειών του Android στις εφαρμογές τρίτων προκειμένου αυτές να αποκτήσουν πρόσβαση στα πιο ευαίσθητα δεδομένα του κινητού ή σε σημαντικούς πόρους του υλισμικού, αποτελεί ένα από τα βασικά μέτρα ασφαλείας του συστήματος. Το εν λόγω σύστημα αναβαθμίστηκε τον Οκτώβριο του 2015, έτσι ώστε ο χρήστης να αποκτήσει μεγαλύτερη επίβλεψη της πρόσβασης αυτής, με την έγκριση της να γίνεται σε πραγματικό χρόνο κατά τη χρήση της εφαρμογής και όχι μόνο στην αρχική εγκατάσταση. Παρά τη βελτίωση αυτή, δεν επιλύθηκαν όλα τα θέματα ασφαλείας της συσκευής και προστασίας της ιδωτικότητας των χρηστών. Συγκεριμένα, η χρήση της διεπαφής του Android API που θα γίνει έπειτα από τους προγραμματιστές εφαρμογών ενέχει νέους κινδύνους και θα πρέπει να έχουν προβλεφτεί μέτρα περιορισμού της. Για το λόγο αυτό η Google που εξελίσσει το Android προβαίνει σε καθημερινό αυτοματοποιημένο έλεγχο εκατοντάδων χιλιάδων εφαρμογών ως προς τη δυνατότητά τους να βλάψουν τις παραπάνω συσκευές για να προστατεύσει τους χρήστες. Στο πλαίσιο της παρούσας εργασίας αναπτύχθηκε με χρήση του Android SDK και της τεχνικής fuzzing η Android εφαρμογή *XenonAutomated*, η οποία χρησιμοποιήθηκε για τον έλεγχο ασφαλείας - κατά πόσο δηλαδή μπορεί μια εξωτερική εφαρμογή να κάνει ‘κακή’ ή ακραία χρήση - των διεπαφών προγραμματιστή Android από το API Level 21 έως το API Level 28. Τα αποτελέσματα που προέκυψαν παρατίθονται για να βρεθούν πιθανά συμπεράσματα.

Abstract

Today, the vast evolution of the technology of smart devices and their operating systems is offering enormous potential for the development of the emerging industry producing applications appropriate for such devices. At the same time, the mobile operating systems become more and more exposed to the danger of malware. The Application Programming Interface provided by **Google’s Android** operating system to software developers undergoes often upgrades and evolves, offering multiple capabilities regarding the exploitation of the device resources. This, in conjunction with the fact that many Android devices are affordable in general, has caused the Android application development market to take off, giving a boost to malicious applications’ evolution too. The permissions system that Android operating system utilizes in order to give third party applications access to user’s sensitive data and vital systems resources of the device, is one of the fundamental security measures of this system. Since October of 2015 it has been upgraded so that the device’s end user has more visibility over that access, by granting the permissions at the point of the actual usage of the respective feature by the applications, instead of once at installation time. However, not all potential security dangers have been eliminated by this enhancement. Moreover, the further usage of the Android API by the application programmers poses new risks, and the platform should be designed to enforce its own proper use. For this reason, in order to protect users, Google for Android uses automated ways for the daily evaluation of hundreds of thousands of applications, regarding their ability to potentially harm the devices. Under the context of this work, the fuzzing test tool *XenonAutomated* was developed and used for an extended security check of possible ‘extreme’ or ‘bad’ usage of the Android API levels from 21 to 28. The results of this test work are presented hereby.

Contents

1. Introduction – Brief Description of the Objective

1.1 Introduction

In modern society, enterprises and individuals rely on smart devices for business operations, collaboration and interpersonal communication, accessing huge amounts of proprietary data and information during their usage. Thus, the integrity of the devices and data protection becomes crucial subject to study.

As the application development business is now mature and part of the world economy [1], users' data are critical and considered valuable. From mobile devices manufactures and developers demonstrating applications specific to such devices, up to mobile advertising networks and affiliate networks [1] taking advantage of multiple channels provided by the mobile device ecosystem in order to generate revenue by digital marketing. It is obvious that with so many stakeholders profiting from mobile devices industry and especially from retrieving users' data, malicious applications become a real danger.

Additionally, users' data produced by the use of smart devices can be really sensitive and proprietary, considering all the evolution of the devices hardware technology. Furthermore, modern smartphones are not anymore simple communicating tools and may be supplied with sensors, the most common of which are the GPS receiver, microphone and camera, while there is a number of other sensors that may be built in, depending on the type and cost of the device, such as barometers, photometers, thermometers, e.t.c. The most popular mobile operating systems provide some middleware, which can be used from third party applications to access all these vital system components mentioned above, and create applications manipulating this private information.

Considering all the above, users need urgently to protect their privacy and avoid any unwanted behavior of their devices, due to harmful applications.

Within this context, the prevailing currently mobile operating systems software companies are engaged intensively in actions towards two directions with conflicting, very often, interests. To offer the tools that enable developers to produce and circulate mobile applications with plenty of capabilities, and at the same time to upgrade their operating systems to defend against potential harmful applications.

Very quickly a new world has been created consisting of three 'components' struggling for equilibrium. The apex of the triangle is companies and independent developers developing thousands of mobile applications daily, by exploiting the new capabilities offered by the devices' and operating systems middleware. At the base of the triangle, on one top are mobiles' and mobiles' operating software enterprises, and on the other top are mobiles' users.

For a number of reasons, the vulnerable component of the triangle consists the users' group, since they constitute valuable sources of information and data, which they cannot assess or manage in order to prevent malware applications from breaching their integrity and privacy. At the same time, mobile users need various applications in order to facilitate everyday life and save time and money. A significant portion of the aforementioned users are in fact plain users, technology illiterate, without any specific knowledge on the potential of such applications and they usually install them with limited awareness of the potential consequences.

Thus, in order to maintain their market share, the enterprises developing mobiles' operating systems software are obliged to constantly upgrade their products to safeguard the interests of both target groups: developers uploading mobile apps, based on their platforms and software tool kits, and the end user. A prominent example is the permissions measure case.

Today, iOS operating system together with the Android platform count for more than 90% of the market share, with the Android platform being the one most popular for mobile devices. The

main reason for this fact is that both operating systems have the option of installation of third party applications. The platform API they provide, which can be used from applications' code to access vital system components of the device, such as sensors, the microphone, the camera, the GPS, SMSs, offer a lot of potential for innovation to mobile developers, but at the same time creates the framework of vulnerability for privacy breaches. Thus, a permissions' system limits the accessibility to the sensitive components, and gives partial control of it to the end user.

The initial model of this permission system, informing users about which device features an application intends to use, and asking for permission once during installation, was dismissed since it was considered insufficient [2]. The problem with it was that application developers started requesting for permissions that neither the end user nor the Application Stores could know or test where will be used, so they could stealthy utilize them at any point, collecting data for several purposes, breaching user's privacy. Subsequently, iOs 7 from Apple initially, and Google later, introduced improvements in their permission models so as the necessary permissions were requested during runtime. Thus, the user is informed about when the application is attempting to handle data or key components (e.g. sensors) and resources of the mobile device and may decide whether to accept or reject the app's intervention, but still use the app (which acts in the interests of the user experience). [2]

However, in order to claim that users' privacy and data are secure, the efficiency of all security models provided by the manufactures should be tested thoroughly. Even if a user grants permission to an application in order to access a critical resource of his device, he still needs to be protected against possible misusage of the resource. Moreover, the latter should be safeguarded by the concerned platform and the platform API.

The Android mobile operating system comprises Google's open source and free software, including middleware and key applications applicable to handset devices. As said above, the Android platform, after its radical redefinition which was implemented by Google, is the one used most widely for both mobile and tablet devices.

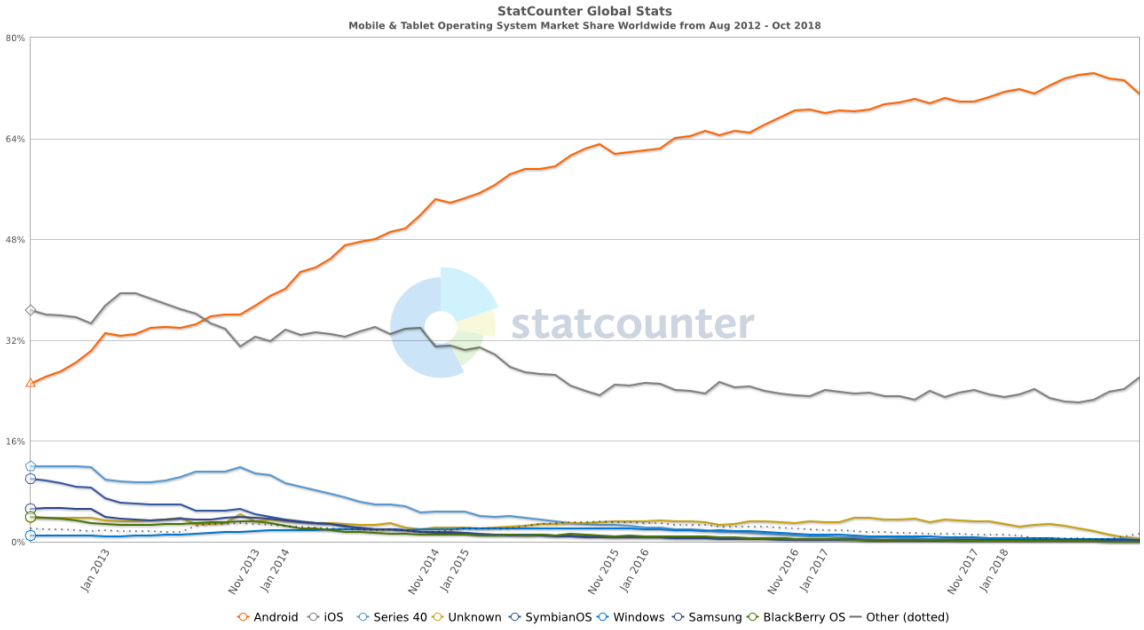

Figure1: http://gs.statcounter.com/os-market-share/mobile-tablet/worldwide/#monthly-201208-201810 [3]

In fact, according to recent statistics (see above Figure 1), Android OS surpassed in about 2013 that time's biggest OS in terms of percentage of mobile/tablet devices having it installed

(that is Apple's iOS), and gradually reached 75% – 85% of the market share in 2016, maintaining its precedence till now. The remaining 15%-20% refers mainly to iOS and a very small percentage is shared by the rest operating systems. [3]

The reason of this tremendous growth is attributed mainly to the fact that device manufactures were allowed to incorporate, often after adaption, the Android's open platform to their products. Thus, there was much flexibility to create and sell several hardware devices equipped with Android at diversified prices, whereas the Apple Company (producing itself both the hardware and the corresponding iOS software) remained at comparatively high prices per unit product.

Considering users' need for low cost efficient and smart mobiles, a consortium consisting of approximately 84 members, including network operators, software developer companies, component and mobile manufacturers and others, *Open Handset Alliance*, came together under the leadership of Google and released the Android Platform, an open-source mobile operating system, unveiling its free distribution in 2007, under the open-source Apache License [4]. The business aliance's goal was to develop open standards for mobile and handset devices, driven by the concept of an open source, free and unified mobile platform that would facilitate low overall cost for handset and development friendly environment that will leverage the post-development. The Android Open Source Project includes the corresponding source code repository which can be used to even edit its code, recompile it and create a custom variant of Android (as it is called, a custom ROM) based on Google's Android platform, replacing the preinstalled firmware of the device manufacturer.

Except the Android core which is released under the open-source Apache License, Android devices use a lot of software which are licensed (Play Store, Google Play Services, Google Music).

Overall, third-party applications for mobile devices are supported by the Android open source platform and application environment. Globally, numerous applications are developed by independent users daily, exploiting the chances that Google Play Store offers. Google Play Store consist of an ideal channel for the distribution and promotion of Android applications.

A key concern is whether security is enforced by the development tools of the SDK against its users - the third-party application developers. As long as these programming tools are generally available for free to the software development community, and Android applications created using them can be freely uploaded to Google Play Store, security measures must be applied prior they are available to the general public. Permissions is one strong security measure, but also the libraries of SDK must be designed properly to serve the security goals. In this context, we are concerned on how someone can misuse the components of this software tool kit in order to harm an Android device and its user's data.

1.2 Main Contributions

The basic contributions of this work can be summarized as follows: We investigate possible vulnerabilities of the Java API framework available by the Android SDK of APIs 21 to 28. Our approach to the problem is to pass 'edge' or ''extreme' values to library classes' methods, to discover unpredictable behavior of the operating system. This has been achieved by building an Android testing application-tool, which utilizes the fuzzing technique and automates the methods invocation with the 'bad' arguments in a full testing suite, but also enables the isolated execution.

1.3 Road Map

The rest of this work is structured as follows: In section 2, we provide brief description of the Android's permission model and present the needs that led to its introduction and some issues recognized as security problems. Then, in section 3, we make a sort reference to the Reflection

concept, in order to elaborate on our attack methodology. In section 4 we introduce fuzzing, which is the software testing technique that we use in this work. In section 5, we present the test application developed to validate our claims, and describe our adversary model, attack methodology and attack evaluation. Also, we cite the results of the tool's execution. Finally, in section 5, we conclude the work.

2. Permissions

Android apps are built to perform a set of actions, some of them requiring permissions from users. Such actions include accessing sensitive user data like contacts and SMSs, or vital hardware features, like the camera and device sensors. Android permissions is a measure for protecting user's privacy. In some cases, a permission might be granted to the application by the system, while in others the user might be prompt to give his consent.

Until Android Lollipop (API level 22), the permission model applied by Google used a "take-it-or-leave-it" approach [2], in which the application permissions should be granted by users right before the application installation gets started. An Android application would retain all of its permissions during its lifecycle, and the only option for revoking a permission would be to uninstall the app. However, a user could not foresee whether an application had good reasons for needing a particular permission or if it would make proper use of it, and there was not much of control over it after the installation. Gradually, it was noticed that developers started adding permissions, which were utilized after the installation with several ways, some of them including data collection, affecting thus the privacy of the user [5]. At that time Google did not check whether or why the app actually required the mentioned permission, while there was no alternative option for the user who wished to continue to use the app, but accepting all permissions at the installation, thus giving developers plenty of time to access system information or the device sensors without any restraint [5].

In October 2015, Google redesigned its permission model with the introduction of Marshmallow (Android version 6.0, API level 23) and concluded with "Runtime Permissions", where users can directly manage application permissions at runtime [6]. According to Google, this new permission model gives to users more control and visibility over the permissions [6], and allows them to selectively repel dangerous permissions in order to adopt the privacy level they want. More specifically, with the "Runtime Permissions" of Google, the developer has to ask during runtime for the permission to perform actions like to use the camera or access the user's storage to save a file, and the user can reject the request but continue using the app.

Additionally, permissions in Android were further classified by Google mainly into normal (see below Table 1) and dangerous (Figure 2 and Table 2). The normal ones, namely those which are not considered quite threatening, are now granted by the system automatically.

Technically, all permissions relate to cases where the application needs to access data or resources outside its sandbox, but some of them, that is the normal ones, are considered to pose lower risk of harming the device, user's data or other applications. Examples of this type include those used for connecting to the internet, setting alarms and wallpapers, and modifying audio settings on a device. On the other hand, potentially dangerous permissions are those giving access to the camera, the microphone and sensors of the device as well as the ones giving access to the contacts' and storage data. In such cases, the user must agree to grant permissions.

As of Android 9 (API level 28), the following permissions are classified as PROTECTION_NORMAL: [7]

Table 1: Normal Permissions - Android 9 (API level 28) [7]

- ACCESS_LOCATION_EXTRA_COMMANDS
- ACCESS_NETWORK_STATE
- ACCESS_NOTIFICATION_POLICY
- ACCESS_WIFI_STATE
- BLUETOOTH
- BLUETOOTH_ADMIN
- BROADCAST_STICKY
- CHANGE_NETWORK_STATE
- CHANGE_WIFI_MULTICAST_STATE
- CHANGE_WIFI_STATE
- DISABLE_KEYGUARD
- EXPAND_STATUS_BAR
- FOREGROUND_SERVICE
- GET_PACKAGE_SIZE
- INSTALL_SHORTCUT
- INTERNET
- KILL_BACKGROUND_PROCESSES
- MANAGE_OWN_CALLS
- MODIFY_AUDIO_SETTINGS
- NFC
- READ_SYNC_SETTINGS
- READ_SYNC_STATS
- RECEIVE_BOOT_COMPLETED
- REORDER_TASKS
- REQUEST_COMPANION_RUN_IN_BACKGROUND
- REQUEST_COMPANION_USE_DATA_IN_BACKGROUND
- REQUEST_DELETE_PACKAGES
- REQUEST_IGNORE_BATTERY_OPTIMIZATIONS
- SET_ALARM
- SET_WALLPAPER
- SET_WALLPAPER_HINTS
- TRANSMIT_IR
- USE_FINGERPRINT
- VIBRATE
- WAKE_LOCK
- WRITE_SYNC_SETTINGS

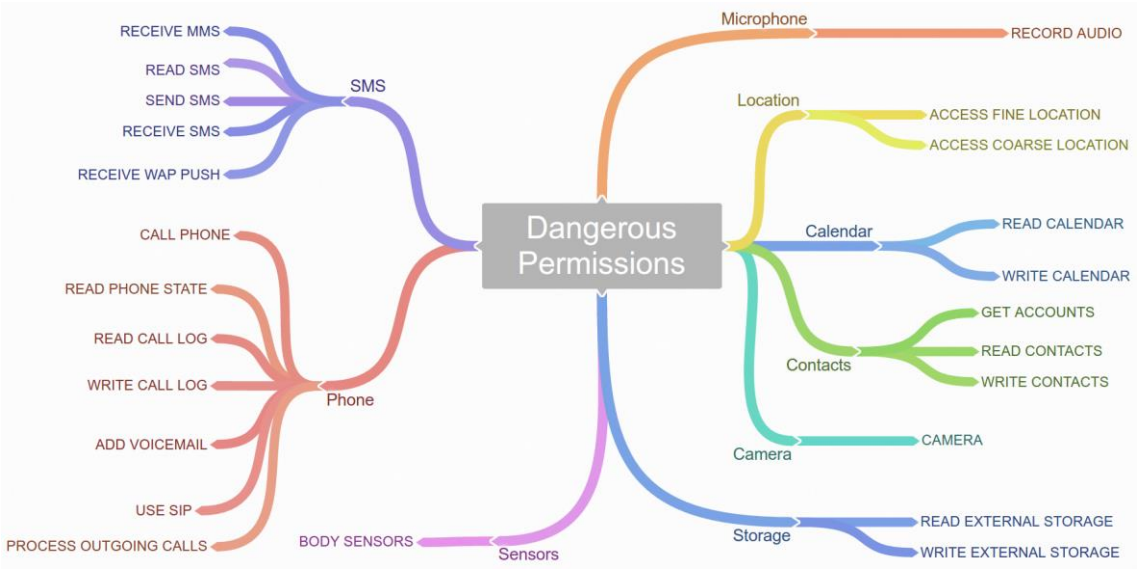

Figure2: Dangerous permissions groups (*Source*: Unravelling Security Issues of Runtime Permissions in Android by Efthimios Alepis and Constantinos Patsakis) [2]

Table 2: Dangerous permissions and permission groups. [7]

Permission Group	Permissions	Permission Group	Permissions
CALENDAR	▪ READ_CALENDAR ▪ WRITE_CALENDAR	MICROPHONE	▪ RECORD_AUDIO
CALL_LOG	▪ READ_CALL_LOG ▪ WRITE_CALL_LOG ▪ PROCESS_OUTGOING_CALLS	PHONE	▪ READ_PHONE_STATE ▪ READ_PHONE_NUMBERS ▪ CALL_PHONE ▪ ANSWER_PHONE_CALLS ▪ ADD_VOICEMAIL ▪ USE_SIP
CAMERA	▪ CAMERA	SENSORS	▪ BODY_SENSORS
CONTACTS	▪ READ_CONTACTS ▪ WRITE_CONTACTS ▪ GET_ACCOUNTS	SMS	▪ SEND_SMS ▪ RECEIVE_SMS ▪ READ_SMS ▪ RECEIVE_WAP_PUSH

			▪ RECEIVE_MMS
LOCATION	▪ ACCESS_FINE_LOCATION ▪ ACCESS_COARSE_LOCATION	STORAGE	▪ READ_EXTERNAL_STORAGE ▪ WRITE_EXTERNAL_STORAGE

Apart from the normal and dangerous, Google has also introduced another protection level, the signature permissions, for Android applications that need to share resources or interoperate with each other. This permission type can only be granted to applications that have been signed with the same certificate as the application that defined the permission. Moreover, signature permissions may refer to either permissions that can be used exclusively by systems apps, or to custom ones, defined by third-party developers. Android platform provides developers the option to define their own permissions as well as permission groups to expose their functionality and data to other apps. Only apps that were developed by the same author and wish to use the same resources can request these defined permissions.

The permissions allowed to be used by third party apps, classified as PROTECTION_SIGNATURE, are as follows for Android 8.1 (API level 27): [7].

Table 3: Protection_Signature Permissions - Android 8.1 (API level 27) [7]

▪ BIND_ACCESSIBILITY_SERVICE	▪ BIND_NOTIFICATION_LISTENER_SERVICE	▪ CLEAR_APP_CACHE
▪ BIND_AUTOFILL_SERVICE	▪ BIND_PRINT_SERVICE	▪ MANAGE_DOCUMENTS
▪ BIND_CARRIER_SERVICES	▪ BIND_SCREENING_SERVICE	▪ READ_VOICEMAIL
▪ BIND_CHOOSER_TARGET_SERVICE	▪ BIND_TELECOM_CONNECTION_SERVICE	▪ REQUEST_INSTALL_PACKAGES
▪ BIND_CONDITION_PROVIDER_SERVICE	▪ BIND_TEXT_SERVICE	▪ SYSTEM_ALERT_WINDOW
▪ BIND_DEVICE_ADMIN	▪ BIND_TV_INPUT	▪ WRITE_SETTINGS
▪ BIND_DREAM_SERVICE	▪ BIND_VISUAL_VOICEMAIL_SERVICE	▪ WRITE_VOICEMAIL
▪ BIND_INCALL_SERVICE	▪ BIND_VOICE_INTERACTION	
▪ BIND_INPUT_METHOD	▪ BIND_VPN_SERVICE	
▪ BIND_MIDI_DEVICE_SERVICE	▪ BIND_VR_LISTENER_SERVICE	
▪ BIND_NFC_SERVICE	▪ BIND_WALLPAPER	

There is another classification of permissions, referred by Google as “special”, as they cannot be classified in any of the other categories. These are the SYSTEM_ALERT_WINDOW and the WRITE_SETTINGS permissions. In fact, a third-party application can actually grant these ones too, after a specific user interaction but it is recommended that applications should not use them and Google documentation strongly discourages developers from requesting them, which implies an increased risk.

Finally, it should be mentioned that the *signatureOrSystem* permission is an old term that corresponds to the "*signature* | *privileged*" permissions (deprecated in API level 23), which is mainly designed for manufactures supplying applications, and they are used in certain specific situations. Such a permission is granted by the system only to apps located in a special folder on the Android system image or that are signed with the same certificate as the application that declared the permission [8]. In case that an app is privileged with such a permission it may have the potential to reboot a device or to clear the caches of all installed applications on the device. [2]

Permissions were further organized into permission groups, which are collections whose members relate to similar functionalities or the same feature of the device. The user will be informed for the permissions requests in group level rather than in permission level. This removed the technical complexity from the previous, probably confusing descriptions that were given to the user. For example, the user will only be asked to grant access to **the contacts** of his device, referring to the CONTACTS group, which includes both READ_CONTACTS and WRITE_CONTACTS permissions.

Regarding dangerous permissions, they all belong to a permission group, while a permission group can consist of any protection's level permission (normal or dangerous) although only the dangerous ones will trigger some user interaction [7]. If an app has already been granted by the user at least one dangerous permission within a specific permission group, then it is automatically granted any other dangerous permission of that group is requested, without additional prompt to the user [7].

To elaborate more on how permissions work in practice, if an application wants to use a system utility, it must include a special xml tag (`<uses-permission>`) with the corresponding permission in the **manifest** file, declaring to the system its intent to use that utility. All kinds of permissions should be mentioned in the manifest file, regardless their protection level or other classification flag, even if the system will treat them differently.

Currently, if an application has declared a target-version tag up to API level 22 **or** the device runs API level up to 22, then all the dangerous permissions for this app will be listed to the user and ask for acceptance in the beginning of the installation process. If the user accept all the permissions, then the app will be granted access to the system's features once and for all, otherwise the installation will be cancelled. In case that an app adds more dangerous permissions in the future, a user that has already installed that app must consent again to all the additional permissions, in order to proceed with the update in his device.

Now, in case a device is running Android API level 23 or higher **and** the declared target-version is Android API level 23 or higher too, then the user doesn't get notified for the dangerous permissions at the installation. The app's developer must include additional implementation, so that the approval happens just before the usage. If so, the user will be prompted to accept the permission at runtime with a special dialog-box. If the user denies the permission, the application will be unable to use the system's feature. The user can revoke his choice at any time, so the application has to be functional with or without the dangerous permissions.

The platform permissions set has not remained the same over the years. Since many unprotected APIs were found in previous versions [2], additional protection mechanisms had been integrated for many intents. Thus, new restrictions were gradually enforced in newer API levels, for actions that were previously allowed by default, and therefor new permissions have been added. For this, Android might automatically add the definitions for the new permissions to an application's manifest file in order to avoid breaking the functionality of older apps installed in newer versions of the API, which assume the free access to the specific APIs [7].

The limitations of the new Permission Model regarding safety have been highlighted in a number of articles and researches. For example, the disclosure of the famous "Cloak and Dagger" class of attacks, driven by scientists of Georgia Institute of Technology [9], proved how a very dangerous application can elicit data like user passwords, or even take control of the device and install other full-permissions-list applications, using the combination of the SYSTEM_ALERT_WINDOW and BIND_ACCESSIBILITY_SERVICE permissions. These two, when used together can create overlays on top of all other applications, with deceptive graphical user interface (often referred as "chat heads"), as it allows malicious activity to be performed underneath, by its hidden parts [10].

Another research at the University of Illinois discovered a trick with which third party applications can gain unauthorized access to system features, by combining custom (declared by untrusted third-party developers) permissions with system permissions [11]. Moreover, it is possible to trick the system to grant the app with the system permissions, by creating an app which initially requests a custom permission with the protection level normal or signature and setting it afterwards to be a part of a system permission group. After installation, these permissions are granted to the app, but an adversary may later update the app to request dangerous permissions which will be automatically granted without any kind of user interaction [2].

The above constitute a quick preview of the main points of the Android permissions' model. As said above, despite the radical changes in security, after the introduction of the new Google Permission Model, the system still presents security vulnerabilities, and the area has yet more room for research.

3. Reflection Overview

The concept of Reflection has been studied in several sciences, like Philosophy, and Logic, but only after a long time of its existence there have been attempts to be formalized [16]. It has been adopted by Artificial Intelligence, as it conceptually represents an ability to perform what we would call an intelligent behavior. It has also found great application in the field of Computer Science, and especially Programming Languages, under the name of computational reflection, originating from Brian Smith's seminal work in the early 80s [13], who introduced two new dialects of the Lisp language, 2-Lisp and 3-Lisp, in his attempt to incorporate the concept of reflection in the area of programming.

A quite general definition of Reflection has been given by Brian Smith during the ECOOP/OOPSLA'90 workshop on reflection as: "*An entity's integral ability to represent, operate on, and otherwise deal with its self in the same way that it represents, operates on and deals with its primary subject matter*." [22].

Reflection has been redefined specifically for programming languages as: "*Reflection is the ability of a program to manipulate as data something representing the state of the program during its own execution. There are two aspects of such manipulation: introspection and intercession. Introspection is the ability for a program to observe and therefore reason about its own state. Intercession is the ability for a program to modify its own execution state or alter its own interpretation or meaning. Both aspects require a mechanism for encoding execution state as data; providing such an encoding is called reification. The parts of the language that support these capabilities are referred to as the introspective and intercessory protocols."* [15]

Reflection in programming languages can be further distinguished in structural and behavioral. When the reification mentioned above relates to the structure of a program, then we refer to structural reflection, while when it refers to the semantics of it, we refer to behavioral reflection. In fact, behavioral reflection is more difficult to be adopted by a programming language as it involves control over the semantics of the language, and thus, structural is incorporated into programming languages in a bigger extend.

Programming languages possessing the abilities of reflection, are languages with embedded self-representations of the language, in the language itself. In fact, languages can support reflective programming, only if they provide a reflective architecture. Along with the introduction of procedural reflection, Smith presented a general framework that enables the addition of the reflective ability to a programming language [17].

In order to add the reflective ability to a programming language, someone must extend the language's model with the model of implementation of the language itself. This means that in a reflective programming language, a program is potentially able to modify its own execution state and semantics during runtime. From a programmer's perspective, it is possible to change the model of the language from within the language [17]. We can notice here that this partially removes the abstraction layer between programmers and the machine accomplished by the programming language, which abstraction consist the initial reason for the creation of programming languages.

Regarding object oriented programming, type introspection constitutes the ability of a program to identify the type and properties of its own objects at runtime. Some OOP languages

go one step further and enable the traversing of the inheritance class tree to discover if an object's class is derived from another. Accordingly, the intercession concept in practice, within the context of OOP programming languages, includes the ability of a program to initialize objects or call methods of a class that were not mentioned at the compilation time, by knowing their class names at runtime, and also to modify the attributes of methods and properties/members of a class definition at runtime.

In Java, reflection allows inspection of classes, interfaces, fields and methods at runtime without knowing the names of the interfaces, fields, methods at compile time. It also allows instantiation of new objects and invocation of methods after the compilation of the program, given their class/method names. In the Java language the reflection feature is implemented through the Reflection API, which is located in the `java.lang.reflect` package, one of the built-in packages that come with Java API. Some important elements of it will be listed below.

A key component for implementing reflection in Java is the class `java.lang.Class`, which models the notion of the Class (models the classes of the language itself). A "class" object can be initialized using its String name, by calling the static `java.lang.Class.forName` method, or by calling the `java.lang.Object.getClass` method of the *Object* class (thus every Java class inherits it). A *Class* object then can be used to introspect every component of the definition of that class (and by class we also include Interfaces, Enums etc) and its elements (methods, fields) as well as their Modifiers (that indicate the encapsulation level).

Regarding methods, the `java.lang.reflect.Method` class of the reflection API can hold a 'method' object retrieved by a call to `java.lang.Class.getMethod`, which can be later on used to actually invoke the respective method, using the `java.lang.reflect.Method.invoke` method. Additionally, someone could get an array with Method objects (corresponding to all methods defined in a class) by calling `java.lang.Class.getDeclaredMethods`.

An object of a specific class can be instantiated using the `java.lang.Class.newInstance` method, or by getting all constructor methods for that class using the `java.lang.Class.getDeclaredConstructors` and then invoking the proper one (the parameters-type list for a constructor would be retrieved by `lang.reflect.Constructor.getParameterTypes`) [18] [19].

Java Reflection API enables a programmer to change the encapsulation level of a field or method at runtime, and convert it from private to public or protected, and so on. This is accomplished by calling the `java.lang.reflect.AccessibleObject.setAccessible` method that both `java.lang.reflect.Method` and `java.lang.reflect.Field` classes inherently have, as they are direct subclasses of the `java.lang.reflect.AccessibleObject` class.

Reflection is a widespread notion in the world of sciences, and a powerful feature in Java that introduces many capabilities to the language which would be impossible otherwise. It has found great application in libraries and tools that need to manipulate objects through xml files or other external sources, and thus, it consists the backbone for many Java frameworks and Integrated Development Environments. The *Spring* framework uses reflection in order to initialize and manage the objects lifecycle through Java beans, and generally implements Dependency Injection through it. *Junit* framework uses reflection to collect through annotations the test methods that need to be invoked. Development environments, like eclipse, need reflection to accomplish auto-complete, and other static analyzing techniques.

However, the Reflection API is also followed by some drawbacks, as it is created to facilitate framework-type of code, giving unlimited access to the language's components, and does not take into consideration the security issues that might raise from it. In cases, for example, where the execution environment is shared with other probably untrusted applications, all this freedom can pose unpredictable risks. A security manager can be applied in such cases, to restrict the accessibility level modification when it is absolutely needed. Additionally, Reflection can cause a

performance hit, when applied to a big extend, as all actions are executed slower through Reflection compared to direct implementation. Finally, the maintainability of the software can be compromised when Reflection is used, as it can obfuscate the code and reduce the clarity and directness of it.

4. Fuzzing

Among the several testing techniques that are used to find security vulnerabilities of software systems, fuzzing methods stand out due to their efficiency and simple logic. In the last years they have gained even more attention, as they accomplish really fast detection of critical security vulnerabilities of applications at certain cases.

Fuzzing (or fuzz testing or fault injection) is a highly automated software testing method with core strategy to provide various invalid, unexpected or random data input to a target software system in order to test it for failures. Moreover, the goal of a "*fuzzer*" testing tool is to provide its target with "semi-valid" input, which is data that are valid enough to avoid getting immediately rejected by their target, while they are invalid enough to cause unexpected behaviors. Fuzzing is mostly used to security vulnerabilities testing.

The concept of fuzzing test is firstly proposed in 1989 by Professor Barton Miller [27] from Wisconsin-Madison College, who developed two programs at his work "An Empirical Study of the Reliability of UNIX Utilities" to test the robustness of the UNIX system, one of which was the *"fuzz". Fuzz* could generate a stream of random characters to be consumed by a target utility program [28].

Fuzzers are often used to test software that accepts structured input, such as specific file formats or protocols, in which cases they can discriminate the valid from invalid input [29]. Additionally, they can test a significantly large number of *boundary cases* quite efficiently. This kind of testing aims to conclude the 'absence' of exploitable vulnerabilities.

Boundary conditions testing is a really big chapter in software testing, especially for systems that accept input values, as they can hide numerus possible failures. Boundary cases can be classified as 'negative test cases', which are meant to prove that an application does **not** do something that is **not** supposed to, in contrast to the 'positive test cases', which verify that the application actually does what it is supposed to (works as designed) and are easier to test using simple functional testing. Thus, for the boundary cases, which are incredibly large in number, the fuzzing methodology has been proven one of the most efficient and fast, as it can give a good code coverage in a short time, due to its automation. It would be impossible for all these cases to be tested with the other known testing techniques.

Another typical usage of fuzzing methodology is for testing the *trust boundary* of applications that accept input from untrusted sources [26], such as a web server that takes input from users. A trust boundary is the point to which some data or the execution moves from one trust level (permissions assigned to resource) to another. For example, when an operating system changes from user to kernel mode, it crosses a trust boundary, as the kernel is trusted to make any of the processor operations, whereas a user's access to them is quite limited [26].

Some common data input points for most systems would include files, APIs, user interfaces, network interfaces, database entries, and command line arguments. Thus, the aforementioned inputs are ideal targets for fuzzing, although anywhere a system can accept any kind of input, malformed data submission should be considered as a possibility. As a general rule, an application that is parsing data entered by a user should be a subject to fuzz testing, so that any possible security vulnerabilities are detected early.

To sum up, a *fuzzer* generates semi-valid data (edge case data that is valid enough to pass through parsers' validations, but as invalid as needed to cause failures), feeds its target system with it, and then observes the system under test to see if it fails during the data consumption. In such case, the tool records the respective submitted data and the observed behavior for later analysis and proceeds with new malformed data [26]. Performing all the above actions manually, the coverage would be incomparably smaller. The fuzzing technique, which automates the entire cycle, can perform even millions of such iterations, covering a significant number of cases, for which it would be difficult to write individual test cases.

We could organize, the whole process of *fuzzing* methodology in five phases: identify the target and the input, generate fuzzing test data, provide the fuzzing test data as input to the target, monitor the failure, determine the utilization [27]. An important step is the test data generation, as the strategy that will be followed here can vary and affect the effectiveness of the testing. Also, it is very important to monitor the application's behavior during a fuzzing test, and debuggers are often used for that purpose. In specific cases, there are other parameters that should be monitored during the fuzzing process too, such as memory usage, network and file system activities etc.

Although fuzz testing has proven successful in revealing critical security vulnerabilities in large applications in short time, traditional *fuzzers* share a well-known common drawback, which is that they can be ineffective if most of their generated malformed inputs are rejected by the target's parsing process in an early stage, and so most test cases fail to access the interior of software (low testing efficiency) [27]. Let's mention here that fuzzing techniques, although being effective in general, cannot be applied to the detection of all kinds of security vulnerabilities that can exist in a software system. Some of the vulnerabilities, due to their nature, are impossible to detect with fuzzing, e.g. when some prior state is required to be initialized before giving the malformed input. The main advantages of the fuzzing technique are simplicity, efficiency and automation, which considerably speeds up the whole process of security vulnerabilities detection.

5. Test Application and Results

5.1 Background

5.1.1 Brief Overview of Android Platform Architecture

In order to elaborate on the Android API's parts that we will test, we should get a bigger picture of the platform structure and the way these APIs communicate with the rest system. In the Figure 4 below, we can see a descriptive diagram of the platform architectonic layers, taken from the official documentation [20], which we will briefly summarize.

The base layer of the Android software stack is the Linux kernel, and the rest system depends on it for low-level operations, such as memory-management and threading. Using the Linux kernel, Android can utilize its UNIX-type security system, which follows a Discretionary Access Control model [23], where each application runs in a different process and as a different user.

A Hardware Abstraction Layer (HAL) on top of the kernel constitutes the communication layer between the higher level Java API framework and the device hardware resources' interfaces of the kernel. It consists of one library module dedicated for each hardware component, and these modules are being loaded whenever the Java API framework requests access to the respective hardware features.

One layer above lays the Android Runtime (ART), as well as the Native C/C++ Libraries. Prior to Android API level 21, the legacy *Dalvik* was in the place of the ART. The ART is designed to execute multiple virtual machines on low-memory devices, by executing a special bytecode format created for Android. This bytecode format is optimized for minimal-memory resources. In devices running Android API level 21 or higher, each app runs in its own process and with its own instance of the ART. ART includes some Android-specific key features, such as optimized Garbage Collection, enhanced debugging support and so on, as well as a set of runtime libraries that contain the most of the functionality of Java Programming Language (including Java 8 features) which are used by the higher-level Java API framework layer. Regarding the Native C/C++ Libraries module, although it facilitates for many of the rest core system components and services, like the ART and the HAL, some of its underling libraries can be accessed directly by native C or C++ code using the Android NDK, and it is used by Android applications developed in C or C++.

The Java API framework contains the entire set of the features provided by the Android operating system, and is available to the outside world (Android programmers) through Java APIs. These APIs can serve as the fundamental building modules for creating Android applications, and include features such as User Interface components (View System), Notification Manager for creating alerts that will be displayed in the device status bar (notifications), as well as an even higher abstraction layer for all the hardware device components featured in HAL, incarnated in these Java APIs (namely the 'Managers'). These APIs written in Java constitute the middleman that will enable apps (system and third-party) to access hardware resources. Our testing app is built on top of that framework (like the other Android applications) which is the one whose security we will actually test.

Finally, the System Apps that come pre-installed to the Android Platform and implement very fundamental device functionalities, such as sending emails or messages and web browsing, comprise the last architectural element of the Android software stack.

5.1.2 Progarmming languages for the Android Platform

Regarding the non-System Android applications, the majority of them are being developed within the Java programming language, the official Android programming language and most supported by Google and Android studio until recently.

However, there are a couple of other languages that someone can also use to create an Android app, such as Kotlin and C/C++ (using the Android NDK as mentioned above). Kotlin, particularly, is very similar to Java language. It is also able to run in the JVM, and is made to interoperate with Java. It must be noted that in October of 2017 Kotlin was announced by Google to be a secondary official Android language, and has been added to the official documentation ever since, which now contains example code-snippets for both Java and Kotlin. However, Java is by far the most popular language for Android development, and the vast majority of the Android apps currently uploaded to Play Store are written in Java.

In general, the Java Development Kit (JDK) is available by the Java language. The Java platform also provides the "Application Programming Interface", or "API" as abbreviation, which is a huge class library appropriate for general-purpose programming. The packages in this library include the Reflection API, which we will use for the exploration, identification and execution of the available components of the Android SDK.

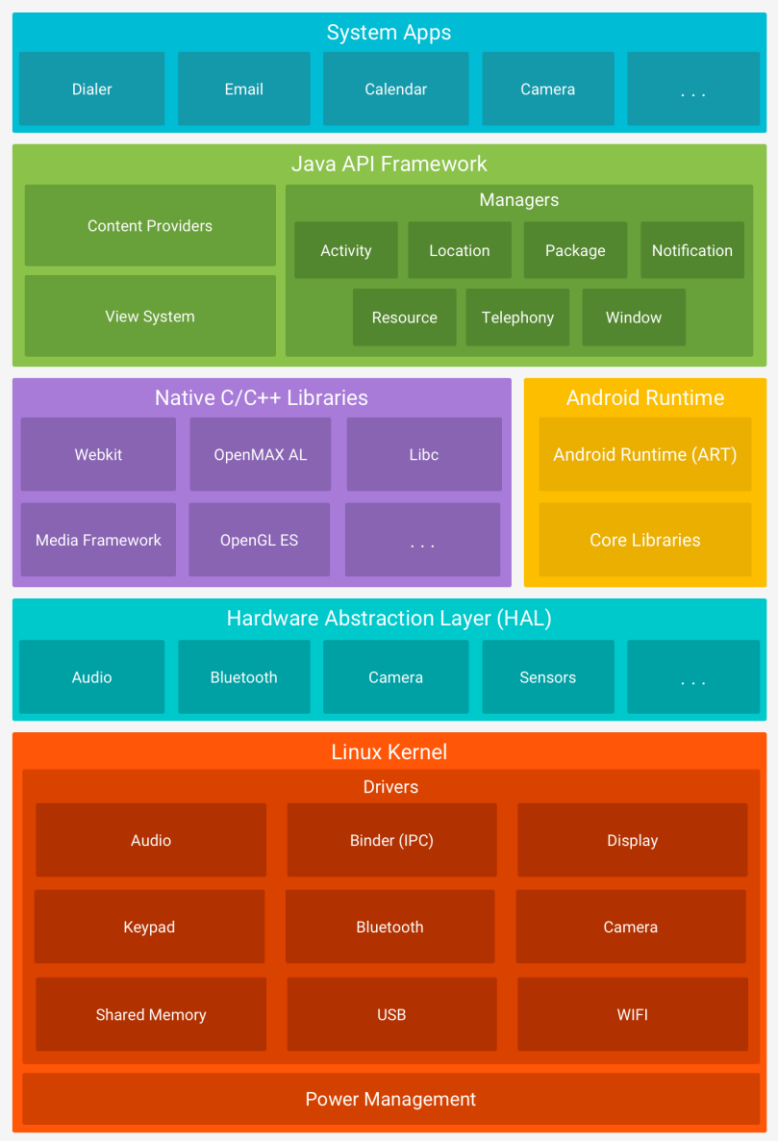

Figure 3: The Android architecture (Source: https://developer.android.com/guide/platform/

5.2 The basic idea

The main idea of our work, is that a malicious Android application will 'exploit' weaknesses of the Android API in order to cause 'harm' to the user of the device installed. The possible harm that is assumed here, would most probably be to compromise the integrity and/or the availability of user's data, or even the device.

5.2.1 The outlining adversary model

The adversary considered in this work, creates the malicious app that a victim installs in her smart device. The app could request for certain permissions, which will facilitate for certain supposed functionality. In such case, the user grants them. During the execution, the app invokes the 'insecure' Android API method with the proper argument values, and causes unexpected reactions to the operating system, such as a restart or UI unresponsiveness. As a result, other

applications running at that time in the device or the file system might experience data loss, because, for example, of an unexpected restart, or even the system might get damaged, which also constitutes data loss.

5.3 The testing tool

5.3.1 The testing environment

This work's testing Android application was written in Java language (for Android). All the development of the application took place in the *Android Studio IDE*. The application was installed and run at some of the available by Android Studio emulator device images, which were created and launched by the *Android Studio's AVD manager tool*. The emulator of our choice was a *Nexus* device. Additionally, the output files of the developed tool ware retrieved from the emulator's file system using the *platform-tools_r28.0.1-windows* tool.

5.3.2 The tool overview

The idea of the testing methodology that *XenonAutomated* application followed was based on the functionality of its ancestor application, *Xenon*. *Xenon* focuses on a category of classes that belong to the Android API (see the Managers in the Java API framework layer of the Android software stack in the diagram above), each one of which is bound to a specific vital hardware component's interface to the operating system, and every method-member of it, facilitates the usage of that component. Such a class has the suffix 'Manager' in its name, as it gives control to its user (that is, the source code of a system application or a third-party application) over the bound hardware component's functionality. For example, the `android.bluetooth.BluetoothManager` class makes available all the functionalities related to the manipulation of the Bluetooth hardware resource of the device.

The main goal of *Xenon* application is to instantiate such a Manager-Class and execute all of its underlying methods with certain 'extreme' arguments passed to the parameter references. These 'extreme' argument values are, in most cases, the maximum and the minimum possible values for the respective type. For example, for Integer parameter types, the Max and the Min integer numerical value supported by the Java language will be passed (e.g. a test method with an Integer parameter would consist of the calls: methodName(Integer.MaxValue); and methodName(Integer.MinValue);). Similarly, for an image parameter, which can be expressed as an `android.graphics.Bitmap` or an `android.graphics.drawable.Icon` object type in our case, the respective Min and Max object arguments would be created using a 'small' and a 'big' - in terms of size in MB - image that is located in the `~Project\app\src\main\res\drawable` package. It can be inferred from the above that, for every method of the class, two invocations will take place. Like in any fuzzing test technique, after each method invocation, a proper "result" will be written in a text file which can be then retrieved from the emulator's file system and opened for study.

An important requirement for the *Xenon* application is to be able to invoke all the methods of a class sequentially but without hardcoded method calls, in order to accomplish the appropriate automation level of the fuzzing. Additionally, this should work for all the Manager-Classes. This is accomplished by the Reflection feature of Java. Moreover, once a Manager-Class is instantiated by the tool, the Reflection API takes over, and creates a Class object for the respective Manager-Class, retrieves the list with its declared methods and creates Method objects for each one. Then, it retrieves the parameter-types array for each Method object, in order to create the corresponding arguments array that should hold the values to be passed. Then, it does the final method invocation. Let's mention here that not actually all methods of a Manager class will be invoked by

our tool; the ones that we are 'interested' in and will be executed are selected according to their parameters list. We have chosen to deal with certain class types consisting the parameters list, which are easy to instantiate with edge values. Furthermore, we started picking methods containing only Integer and String type of parameters, and gradually – in the context of the extended *XenonAutomated* application – also added Float, Double, Boolean, CharSequence, Long, Sort, byte, char, Icon, Bitmap and Context parameter types in the list (for all primitive types of java, both the equivalent Class type and the primitive type are included).

This is, more or less, how *Xenon* application does the methods invocation. The last part to complete the presentation of *Xenon* functionality is how the Manager-Classes are instantiated in the first place. This is accomplished with the help of the fundamental **Context** class of the Android API. Furthermore, the `android.content.Context` Java abstract class of the Android SDK constitutes the basic access point to the system features interfaces' implementation, and is responsible for providing instances for all the Manager-Classes. The Activity class, as the building block for an Android application, extends the Context class, thus an initialized Context instance reference is always available from within an Activity (or a subclass of the Activity) by a call to `this`. Context class definition contains a set of String constant members, each one of which corresponds to a Manager-Class and should be used in order to retrieve an instance of the respective Manager. For example, the `android.content.Context.getSystemService(BATTERY_SERVICE)` method call will return back an instance of the `BatteryManager` Class, which is able to handle any functionalities for the battery hardware component [24].

The execution of our fuzzer tool should be monitored. While executing all the methods for a Manager-class, someone shall observe the emulator's behavior, as well as the Log output of the Android Studio IDE, and notice if anything unexpected happens. The conclusions will come from the output file, as well as the observations.

The enhancements that was needed to be added to the tool were, on the one hand, the app's ability to execute all these classes previously discussed sequentially, without any additional manual action, as the existing implementation could only run one class in a single execution, and, on the other hand, to keep track of the execution progress so that it continues executing from the next method or class in case of a crash. In general, it should keep a 'state' of the execution in a persistent storage. This state should be aware of the list of classes and methods that remain for a full execution to complete.

The requirement for these new features led to the *XenonAutomated* creation, which retains the core testing implementation (as described above) unchanged, but introduces this new state awareness functionality, that orchestrates the execution of the whole class list in a full testing suite. This speeds up significantly the investigation pace towards the direction we want, and increases the efficiency of the fuzzing method implemented. Once the application crashes, then it can be re-run and the execution will continue from the next declared method of the class, or the next class of the list when the method list empties. Additionally, the output of the execution is written in an Excel file, one row for each method. If the invoked method returns normally, a simple 'SUCCESS' message will assigned to the result, whereas if it throws an exception that can be caught, the exception message will be assigned to the result. Finally, if a thrown exception does not get caught and causes the application to crash, probably nothing will be written in the column of the result, and the application should be restarted by the tester that leads the investigation. However, if the method invocation causes not only the application, but the whole system to crash (which is the main case under investigation), like an OS restart, this should be observed by the tester (and again, nothing will be written to the result column at the Excel report). Then, this observation should be written down to a report file. Note that, the result columns will include both the minimum and the maximum values' invocation result.

Another extra functionality added to the extended app is the dynamic retrieval of the classes list. After the addition of the execution-state storage in the app, the classes to be executed had to be stored in a persistent list too. And, by saying 'classes' it is meant the constant fields that shall be used to request from Context the relative class object, just as described above. In the previous version of the app, the class was always one, and the constant was hardcoded in the program (and the hardcoded value could be changed on demand). For the extended version, it was considered preferable the constant values to be dynamically resolved during runtime, from a credible source. The retrieval is accomplished by calling an HTTP request to the documentation website, and parsing over the result.

This implementation gave the idea of further expansion of the range of the classes under test, which could include even more classes of the Android API. Thus, additional modules were added that acquire lists of Android SDK API classes (class names) from the official documentation. For these extra Android API classes the object instantiation is accomplished differently, as they are not provided by the Context class (these new classes are not necessarily related to the hardware components; they are of general purpose). Instead, the solution was provided by the Reflection API, once again. Using the class name, a 'Class' type object can be created for that class. Then, all constructors declared for that class can be returned by the Reflection API, and a proper one can be chosen to initialize the object (method invocation by the Reflection API). The rest process remains the same.

In like matter, another expansion idea that came up naturally was to convert all methods to public, with the use of Reflection. After executing some classes, it was observed that some methods were throwing illegal-accessibility exception messages, as they were declared as protected or private in their class definition, and the successful invocation was deprived by the language. Reflection provides this feature of intercession, that enables the modification of the accessibility level of a method (or a member), and this was used by this work in order to turn all method modifiers to public, and then do the invocation, therefore bypassing the `IllegalAccessException`.

During the development of *XenonAutomated* app, it was found that it is useful to be able to isolate a single method of a class and execute it independently, in case, for example, of a strange behavior of the operating system where we want to identify which exactly is the root cause. As such, we added the additional feature of two drop down lists: one containing the full classes list, and the second dynamically rendered upon list item selection of the first one, containing all the methods of the selected class (see the screenshots shown in the figures 4, 5 below). Thus, someone can invoke the selected method only. With this functionality, along with the use of the debugging support of the Android Studio, someone can focus on specific methods testing, and make conclusions about the observed behaviors.

5.3.3 The tool and Android permissions.

According to the initial idea of this work, regardless of the permissions granted by the user to the application, malicious actions could be performed, exploiting Android API's vulnerabilities. As such, we have implemented the tool so that is requests **all** the available to third parties permissions of the current API level running on the device (and the target version of the tool). The user of the tool must approve each one, granting all permissions for the app. This set of user interactions will happen at the beginning, after the first launch of the application, so that when the testing begins, all possible Android APIs are accessible. In the adversary model this tool is aiming, an appropriate subset of these permissions would be used - the ones needed to access a vulnerable API component. The unsuspecting user, would allow an application to use some of the vital system components, but nevertheless, he would not expect to experience some damage in his device, and in general, granting permissions does not imply allowing harming operations to act on the device.

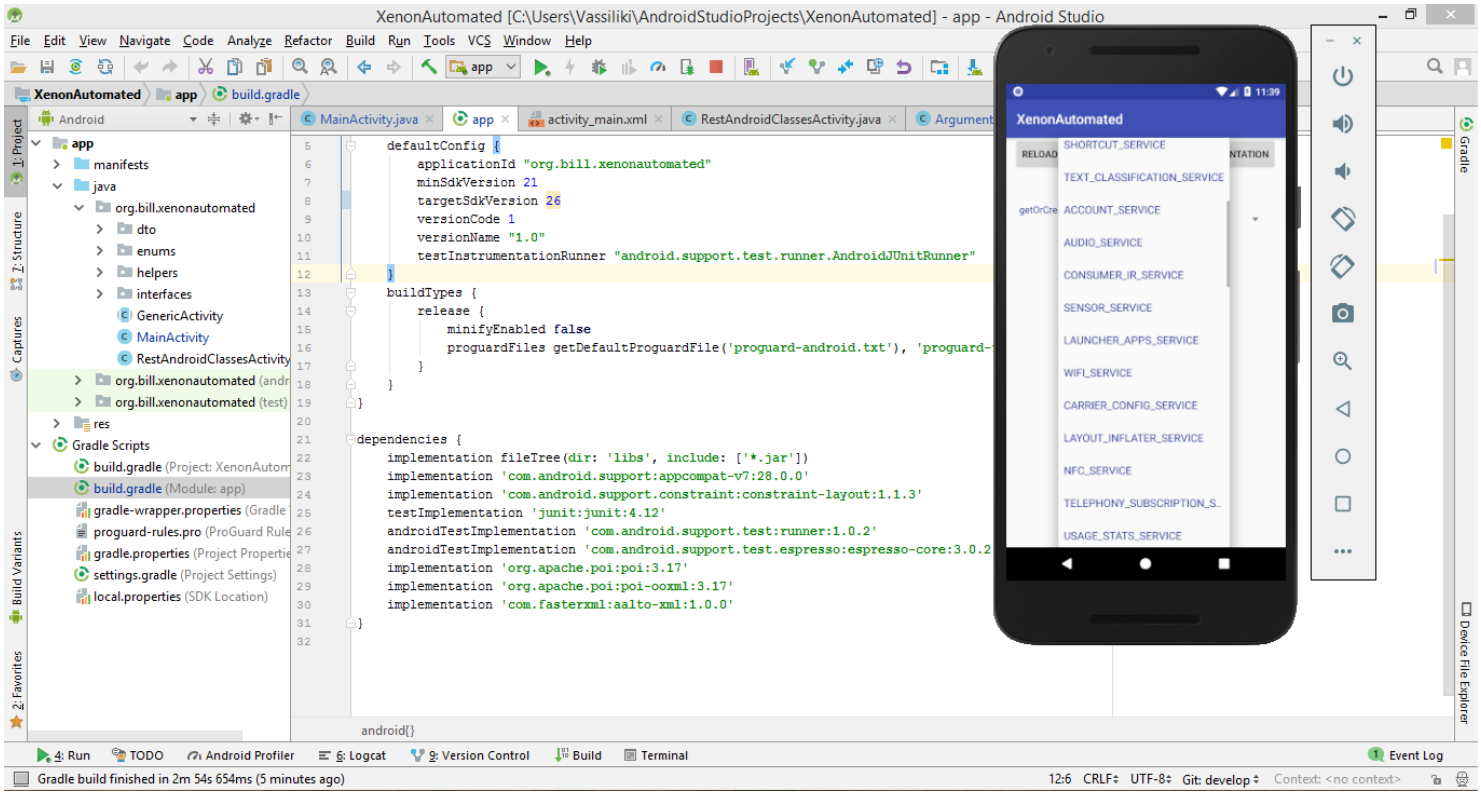

Figure 4 : The *XenonAutomated* application Tool running in the Emulator of the Android Studio

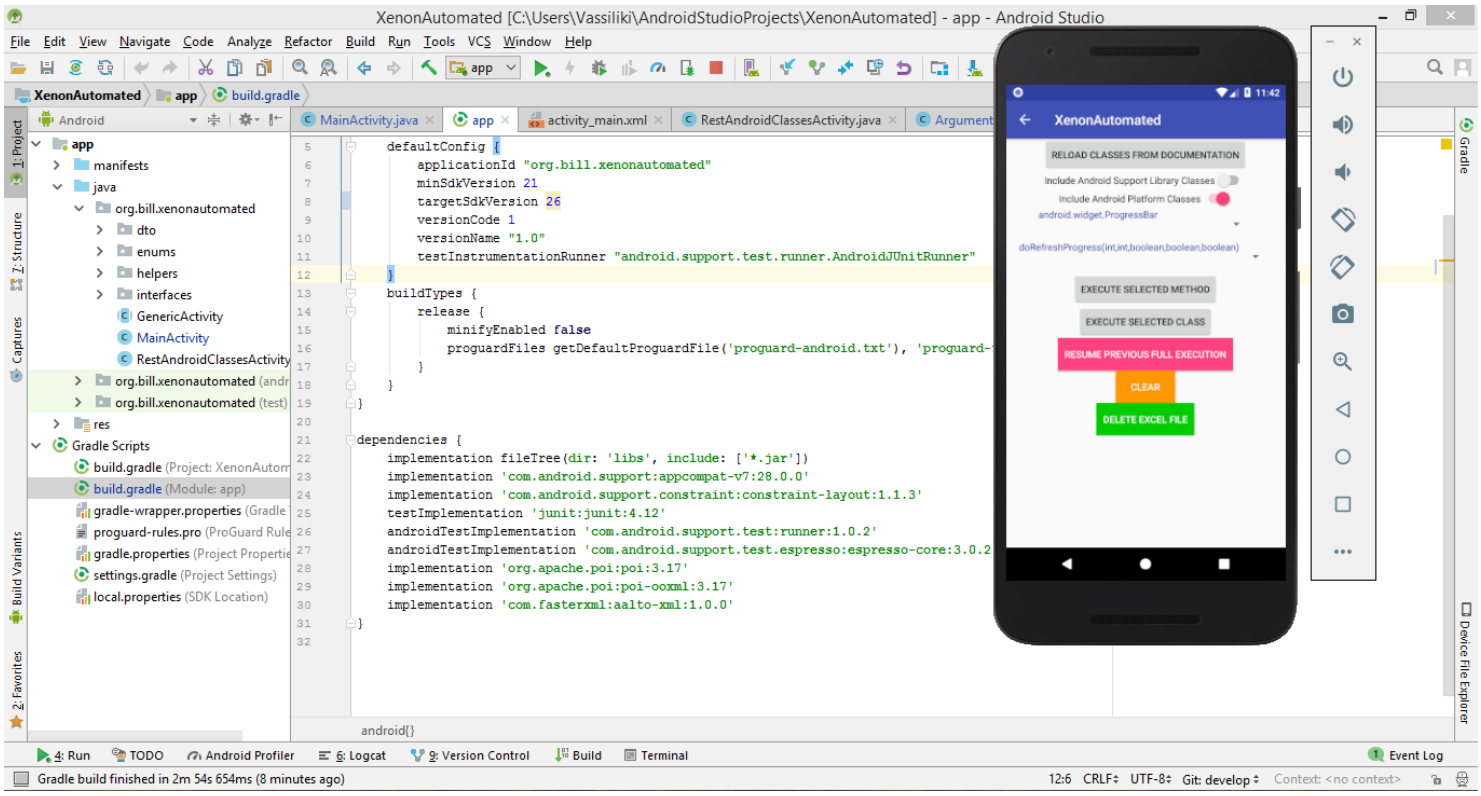

Figure 5: The *XenonAutomated* application Tool running in the Emulator of the Android Studio

5.3.4 Tool architecture

With respect to the code structure, *XenonAutomated* is a relatively simple Android application, in order to facilitate its goal better. It consists of two Activities (accordingly two application views), which split the functionality in two parts according to the classes they test. These two inherit from a generic activity class, as there is a lot of common implementation between them. The several services that help the activities implement the logic, are spread to sub-packages in order to decompose the functionality to smaller entities. A key feature of the implementation is the methodology by which the arguments passed to the methods under test are created. The latter is centralized for all the places where we have test-method invocation (e.g. for the single method execution, or the full class execution). This is accomplished with the help of an *Enumeration* class that declares all the possible parameter types, and a map that is available to all the activities, which maps an *Enumeration* value to a lambda (callback) that will return (when the under-test method is called) the corresponding min and max values (see code-snippets below). This design enables the extensibility of the tool, as even more 'accepted' class types can be added to the list in the future, with minimum effort. It also facilitates the flexibility to change the values provided to the methods on demand, making it more D.R.Y. (the developer does not have to search in many places to do the same work).

```
public enum ValidArguments {
    INT("int"),
    INTEGER("java.lang.Integer"),
    RESOURCE_ID("resource.id"),
    LONG("long"),
    LONG_CLASS("java.lang.Long"),
    STRING("java.lang.String"),
    BOOLEAN("boolean"),
    BOOLEAN_CLASS("java.lang.Boolean"),
    CHAR_SEQUENCE("java.lang.CharSequence"),
    FLOAT("float"),
    FLOAT_CLASS("java.lang.Float"),
    DOUBLE("double"),
    DOUBLE_CLASS("java.lang.Double"),
    SHORT("short"),
    SHORT_CLASS("java.lang.Short"),
    BYTE("byte"),
    CHAR("char"),
    ICON("android.graphics.drawable.Icon"),
    BITMAP("android.graphics.Bitmap"),
    CONTEXT("android.content.Context");

    private final String value;
    ValidArguments(String val) { this.value = val; }
    public String getValue() { return this.value; }
```

Figure 6: Code snippet of The *XenonAutomated* application Tool

```
public class ArgumentsCreator extends HashMap<ValidArguments,MinMaxArhumentCreator>{

    public void initializeArgumentCreator(Context context) {
        addAllMappings(context);
    }
    private void addAllMappings(final Context context)
    {
        /* Initialize map */
        this.put(ValidArguments.INT,new MinMaxArhumentCreator(() → {
                return Integer.MAX_VALUE;
        }, () → { return Integer.MIN_VALUE; },() → { return 1; }));
        this.put(ValidArguments.INTEGER,new MinMaxArhumentCreator(() → {
            return Integer.MAX_VALUE;
        }, () → { return Integer.MIN_VALUE; },() → { return 1; }));
        this.put(ValidArguments.LONG,new MinMaxArhumentCreator(() → {
                return Long.MAX_VALUE;
        }, () → { return Long.MIN_VALUE; },() → { return 112L; }));
        this.put(ValidArguments.LONG_CLASS,new MinMaxArhumentCreator(() → {
                return Long.MAX_VALUE;
        }, () → { return Long.MIN_VALUE; },() → { return 112L; }));
        this.put(ValidArguments.SHORT,new MinMaxArhumentCreator(() → {
                return Short.MAX_VALUE;
        }, () → { return Short.MIN_VALUE; },() → { return 5; }));
        this.put(ValidArguments.SHORT_CLASS,new MinMaxArhumentCreator(() → {
                return Short.MAX_VALUE;
        }, () → { return Short.MIN_VALUE; },() → { return 5; }));
        this.put(ValidArguments.FLOAT,new MinMaxArhumentCreator(() → {
                return Float.MAX_VALUE;
        }, () → { return Float.MIN_VALUE; },() → { return 3.6f; }));
        this.put(ValidArguments.FLOAT_CLASS,new MinMaxArhumentCreator(() → {
                return Float.MAX_VALUE;
```

Figure 7: Code snippet of The *XenonAutomated* application Tool

5.4 Results

After completing the execution of *XenonAutomated* app for the APIs 21 to 28 inclusive, we can describe the output of the testing.

We tested numerous methods, belonging to over 180 classes of the Android API. A finding is that there were some method invocations that caused intervention to the mobile's UI, which can be considered as valid for some of these cases. However, for some other cases, we can question how proper such actions are, when performed by third-party applications. For example, in the case of android.telephony.TelephonyManager, the dial method opened a telephony-manager view and typed several numbers of the keyboard, while the application kept running in the background. The user is supposed to have granted all telephony-related permissions to the app, but typing numbers on the keyboard of a telephony-manager Android view, almost attempting to perform a phone call, cannot be considered as a proper use of the aforementioned permissions. Additionally, the ability to dial a phone number is not even documented as an available action.

The most outstanding outcome of the test results is arguably the cases where the method invocation caused a restart to the Operating System. Moreover, this behavior was observed at four methods in total, and they all concern Android API level 26: the `android.media.AudioManager.adjustSuggestedStreamVolume(int,int,int)`, the `android.media.AudioManager.adjustVolume(int,int)`, the

`android.media.AudioManager.setStreamMute(int,boolean)` under certain circumstances and the `android.media.session.MediaSessionManager.dispatchAdjustVolume(int,int,int)`. The above methods were executed isolated in debugging mode in order to be tested thoroughly, as we should confirm the origin of the system failure. The finding is that right after the method call, the operating system rebooted. This even more dangerous 'unpredictable' behavior could even be combined with other security currently unresolved issues of the platform, to seriously harm the user.

A sample of the most important result tables (coming out of the extracted Excel files), are presented below. The first table is an example of the output (originally Excel) file that our tool produces. After that, we presented an overview table containing the observations for all the Context classes methods, for all APIs under test. There were many more classes investigated, with no significant results to present.

Table 4: Example sheet of the code running_API_26 (part of the excel sheet)

Class Name	Method Name	Arguments List	Invoke Result Min	Invoke Result Max
android.hardware.display.DisplayManager	getOrCreateDisplayLocked	(int , boolean)	SUCCESS	SUCCESS
android.hardware.display.DisplayManager	connectWifiDisplay	(java.lang.String)	InvocationTargetException: null Cause: Permission required to connect to a wifi display: Neither user 10085 nor current process has android.permission.CONFIGURE_WIFI_DISPLAY .	InvocationTargetException: null Cause: android.os.TransactionTooLargeException: data parcel size 42949772 bytes
android.hardware.display.DisplayManager	forgetWifiDisplay	(java.lang.String)	InvocationTargetException: null Cause: Permission required to forget to a wifi display: Neither user 10085 nor current process has android.permission.CONFIGURE_WIFI_DISPLAY .	InvocationTargetException: null Cause: android.os.TransactionTooLargeException: data parcel size 42949772 bytes
android.hardware.display.DisplayManager	getDisplay	(int)	SUCCESS	SUCCESS
android.hardware.display.DisplayManager	getDisplays	(java.lang.String)	SUCCESS	SUCCESS
android.hardware.display.DisplayManager	renameWifiDisplay	(java.lang.String , java.lang.String)	InvocationTargetException: null Cause: Permission required to rename to a wifi display: Neither user 10085 nor current process has android.permission.CONFIGURE_WIFI_DISPLAY .	InvocationTargetException: null Cause: android.os.TransactionTooLargeException: data parcel size 85899452 bytes

Class Name	Method Name	Arguments List	Invoke Result Min	Invoke Result Max
android.hardware.fingerprint.FingerprintManager	getAcquiredString	(int , int)	SUCCESS	SUCCESS
android.hardware.fingerprint.FingerprintManager	getErrorString	(int , int)	SUCCESS	SUCCESS
android.hardware.fingerprint.FingerprintManager	getEnrolledFingerprints	(int)	SUCCESS	SUCCESS
android.hardware.fingerprint.FingerprintManager	hasEnrolledFingerprints	(int)	InvocationTargetException: null Cause: Must have android.permission.INTERACT_ACROSS_USERS permission.: Neither user 10085 nor current process has android.permission.INTERACT_ACROSS_USERS.	InvocationTargetException: null Cause: Must have android.permission.INTERACT_ACROSS_USERS permission.: Neither user 10085 nor current process has android.permission.INTERACT_ACROSS_USERS.
android.hardware.fingerprint.FingerprintManager	rename	(int , int , java.lang.String)	InvocationTargetException: null Cause: Must have android.permission.MANAGE_FINGERPRINT permission.: Neither user 10085 nor current process has android.permission.MANAGE_FINGERPRINT.	InvocationTargetException: null Cause: android.os.TransactionTooLargeException: data parcel size 42949796 bytes
android.hardware.fingerprint.FingerprintManager	setActiveUser	(int)	InvocationTargetException: null Cause: Must have android.permission.MANAGE_FINGERPRINT permission.: Neither user 10085 nor current process has android.permission.MANAGE_FINGERPRINT.	InvocationTargetException: null Cause: Must have android.permission.MANAGE_FINGERPRINT permission.: Neither user 10085 nor current process has android.permission.MANAGE_FINGERPRINT.

Class Name	Method Name	Arguments List	Invoke Result Min	Invoke Result Max
android.view.inputmethod.InputMethodManager	checkFocusNoStartInput	(boolean)	SUCCESS	SUCCESS
android.view.inputmethod.InputMethodManager	finishedInputEvent	(int , boolean , boolean)	SUCCESS	SUCCESS
android.view.inputmethod.InputMethodManager	setUpdateCursorAnchorInfoMode	(int)	SUCCESS	SUCCESS
android.view.inputmethod.InputMethodManager	showInputMethodAndSubtypeEnabler	(java.lang.String)	SUCCESS	InvocationTargetException: null Cause: android.os.TransactionTooLargeException: data parcel size 42949796 bytes
android.view.inputmethod.InputMethodManager	showInputMethodPicker	(boolean)	SUCCESS	SUCCESS
android.view.inputmethod.InputMethodManager	toggleSoftInput	(int , int)	SUCCESS	SUCCESS
android.app.UiModeManager	disableCarMode	(int)	SUCCESS	SUCCESS
android.app.UiModeManager	enableCarMode	(int)	SUCCESS	SUCCESS
android.app.UiModeManager	setNightMode	(int)	InvocationTargetException: null Cause: Unknown mode: -2147483648	InvocationTargetException: null Cause: Unknown mode: 2147483647
android.telecom.TelecomManager	from	(android.content.Context)	SUCCESS	SUCCESS
android.telecom.TelecomManager	acceptRingingCall	(int)	SUCCESS	SUCCESS

Class Name	Method Name	Arguments List	Invoke Result Min	Invoke Result Max
android.telecom.TelecomManager	clearAccountsForPackage	(java.lang.String)	InvocationTargetException: null Cause: Package b does not belong to 10085	SUCCESS
android.telecom.TelecomManager	getCallCapablePhoneAccounts	(boolean)	SUCCESS	SUCCESS
android.telecom.TelecomManager	getDefaultOutgoingPhoneAccount	(java.lang.String)	SUCCESS	SUCCESS
android.telecom.TelecomManager	getPhoneAccountsSupportingScheme	(java.lang.String)	SUCCESS	SUCCESS
android.telecom.TelecomManager	getSimCallManager	(int)	SUCCESS	SUCCESS

Table 5: Results after running the Xenon Code - API_26

Class Name	Result
android.hardware.display.DisplayManager	No changes in the OS
android.hardware.fingerprint.FingerprintManager	No changes in the OS
android.view.inputmethod.InputMethodManager	showInputMethodAndSubtypeEnabler (java.lang.String) -> white view poped up, application kept running in the background toggleSoftInput (int , int) -> keyboard poped up and application kept running normally, showInputMethodPicker(boolean) -> a language picker poped up and application kept running normally
android.app.UiModeManager	disableCarMode(int) -> application closed down but kept running in the background enableCarMode(int) -> black view poped up on top of the application.
android.telecom.TelecomManager	No changes in the OS
android.app.admin.DevicePolicyManager	No changes in the OS
android.view.accessibility.AccessibilityManager	No changes in the OS
android.app.AppOpsManager	No changes in the OS
android.hardware.camera2.CameraManager	No changes in the OS
android.content.pm.ShortcutManager	No changes in the OS
android.view.textclassifier.TextClassificationManager	No changes in the OS
android.accounts.AccountManager	No changes in the OS
android.media.AudioManager	adjustSuggestedStreamVolume(int,int,int) -> OS restart, adjustVolume(int,int), setStreamMute(int ,boolean)-> OS restart under certain conditions
android.media.session.MediaSessionManager	dispatchAdjustVolume(int ,int ,int)-> OS restart
ERROR-->CLASS: TV_INPUT_SERVICE	ERROR
android.view.textservice.TextServicesManager	No changes in the OS
android.app.AlarmManager	No changes in the OS
android.app.usage.NetworkStatsManager	No changes in the OS
android.app.KeyguardManager	No changes in the OS
android.app.SearchManager	No changes in the OS
android.companion.CompanionDeviceManager	No changes in the OS
android.os.UserManager	No changes in the OS
android.appwidget.AppWidgetManager	No changes in the OS
android.app.NotificationManager	No changes in the OS

Class Name	Result
android.app.DownloadManager	No changes in the OS
android.view.WindowManagerImpl	No changes in the OS
android.hardware.SystemSensorManager	No changes in the OS
android.content.pm.LauncherApps	No changes in the OS
android.app.ActivityManager	No changes in the OS
android.net.wifi.WifiManager	No changes in the OS
android.net.ConnectivityManager	No changes in the OS
android.telephony.CarrierConfigManager	No changes in the OS
com.android.internal.policy.PhoneLayoutInflater	No changes in the OS
android.os.PowerManager	No changes in the OS
android.hardware.input.InputManager	No changes in the OS
android.media.MediaRouter	No changes in the OS
android.content.ClipboardManager	No changes in the OS
android.telephony.SubscriptionManager	No changes in the OS
android.view.accessibility.CaptioningManager	No changes in the OS
android.telephony.TelephonyManager	dial(java.lang.String) ->A phone manager view with keyboard appeard, and 2 was typed. Appplication continued running in the background.
android.bluetooth.BluetoothManager	No changes in the OS
android.os.storage.StorageManager	No changes in the OS
android.app.usage.UsageStatsManager	No changes in the OS
android.location.LocationManager	No changes in the OS
android.os.HardwarePropertiesManager	No changes in the OS
android.app.JobSchedulerImpl	No changes in the OS
android.content.RestrictionsManager	No changes in the OS
android.os.health.SystemHealthManager	No changes in the OS
android.app.usage.StorageStatsManager	No changes in the OS
android.os.DropBoxManager	No changes in the OS
android.hardware.usb.UsbManager	No changes in the OS
ERROR-->CLASS: WIFI_AWARE_SERVICE	ERROR
ERROR-->CLASS: WIFI_P2P_SERVICE	ERROR
android.os.BatteryManager	No changes in the OS
android.net.nsd.NsdManager	No changes in the OS

Table 6: Results after running the Xenon Code - API_21

Class Name	Result
android.view.inputmethod.InputMethodManager	toggleSoftInput (int , int) -> keyboard poped up and application kept running normally
android.app.UiModeManager	disableCarMode(int) -> application closed down but kept running in the background.

Table 7: Results after running the Xenon Code - API_22

Class Name	Result
android.telephony.TelephonyManager	dial(java.lang.String) ->A phone manager view with keyboard appeard, and 2 was typed. Appplication continued running in the background.
android.app.UiModeManager	disableCarMode(int) -> application closed down but kept running in the background
android.view.inputmethod.InputMethodManager	toggleSoftInput (int , int) -> keyboard poped up and application kept running normally

Table 8: Results after running the Xenon Code - API_23

Class Name	Result
android.view.inputmethod.InputMethodManager	toggleSoftInput (int , int) -> keyboard poped up and application kept running normally, showInputMethodPicker(boolean) -> a language picker poped up and application kept running normally.
android.telephony.TelephonyManager	dial(java.lang.String) ->A phone manager view with keyboard appeard, and 2 was typed. Appplication continued running in the background.
android.app.UiModeManager	disableCarMode(int) -> application closed down but kept running in the background

Table 9: Results after running the Xenon Code - API_24

Class Name	Result
android.telephony.TelephonyManager	dial(java.lang.String) ->A phone manager view with keyboard appeard, and 2 was typed. Appplication continued running in the background.
android.view.inputmethod.InputMethodManager	showInputMethodAndSubtypeEnabler (java.lang.String) -> white view poped up, application kept running in the background toggleSoftInput (int , int) -> keyboard poped up and application kept running normally, showInputMethodPicker(boolean) -> a language picker poped up and application kept running normally
android.app.UiModeManager	disableCarMode(int) -> application closed down but kept running in the background

Table 10: Results after running the Xenon Code - API_25

Class Name	Result
android.os.BatteryManager	dial(java.lang.String) ->A phone manager view with keyboard appeard, and 2 was typed. Appplication continued running in the background.
android.view.inputmethod.InputMethodManager	showInputMethodAndSubtypeEnabler (java.lang.String) -> white view poped up, application kept running in the background toggleSoftInput (int , int) -> keyboard poped up and application kept running normally, showInputMethodPicker(boolean) -> a language picker poped up and application kept running normally
android.app.UiModeManager	disableCarMode(int) -> application closed down but kept running in the background

Table 11: Results after running the Xenon Code - API_27

Class Name	Result
android.view.inputmethod.InputMethodManager	showInputMethodAndSubtypeEnabler (java.lang.String) -> white view poped up, application kept running in the background toggleSoftInput (int , int) -> keyboard poped up and application kept running normally, showInputMethodPicker(boolean) -> a language picker poped up and application kept running normally
android.app.UiModeManager	disableCarMode(int) -> application closed down but kept running in the background enableCarMode(int) -> black view poped up on top of the application.
android.telephony.TelephonyManager	dial(java.lang.String) ->A phone manager view with keyboard appeard, and 2 was typed. Appplication continued running in the background.

Table 12: Results after running the Xenon Code - API_28

Class Name	Result
android.view.inputmethod.InputMethodManager	showInputMethodAndSubtypeEnabler -> white view poped up, application kept running in the background toggleSoftInput -> keyboard poped up
android.app.UiModeManager	disableCarMode -> application closed down but kept running in the background enableCarMode -> black view poped up on top of the application.
android.telephony.TelephonyManager	dial ->A dial view with keyboard appeard, and 2 was typed. Appplication continued running in the background.

6. Summary - Conclusions

In this work, we introduced the reader to general concerns regarding the Android ecosystem that led us search specific security breaches of the Android platform API. We outlined the Android permissions model and its historical evolution, as well as some of its reported, at various times, vulnerabilities, as it constitutes a fundamental security layer for the Android system. We briefly summarized the concept of reflection in science and its incarnation to the Java language, as it is a key feature used in the development of is work. We introduced the "fuzzing" technique, which is a software testing methodology mostly used to reveal security vulnerabilities, as is the one used by out tool. We presented the adversary model considered for this work, and outlined the platform components we indented to test. Finally, we gave analytical overview of the tool developed to validate our claims (*XenonAutomated*), and reported sample of the most interesting results of the tool's execution.

The claim considered for this work is that permissions security measure alone cannot provide full protection over the device and data integrity, privacy and availability, as their approval should allow limited and safeguarded access to the corresponding APIs. The testing methodology that was adopted is to pass some edge case values to the methods' parameters and check for unpredictable behavior of the system (fuzzing). The execution was performed over numerous methods of each Android API - from level 21 to level 28 - and showed some remarkable UI interventions and a few methods in the Android API level 26 that can potentially harm Android user, since they cause operating system's restart.

In this work, the parameters list of the API methods that were tested included specific types of arguments. This module of the tool is quite extensible, as the list can be enriched in a future work, and even more methods can be put under the microscope. Further investigation on the matter that would spread to more classes and methods, could potentially disclosure more interesting results. Another test parameter that could be modified in the future in order to increase the tool's effectiveness, is the algorithm used by our *fuzzer*. A number of algorithms have been adopted by *fuzzers* over the years for randomizing or distributing the data input appropriately, and similar logic can be adopted by this tool for experimentation.

We also concluded that the limitations of the fuzzing technique in general, restricted the results of this work. In some cases, the methods' execution should have been performed in a 'logical' sequence, in a way that makes sense and aims to a particular action, so that a prior 'state' has been initialized, in order to unravel the hidden vulnerabilities. Additionally, it was noticed that some of the edge values passed to the API methods caused an early exception related to 'size' validations, and did not manage to pass to the interior system and cause a failure. After all, this is the most common weakness of the fuzzing technique that can possibly affect the effectiveness of the testing.

7. Acknowledgments

The first version of the application developed for this work, *Xenon*, which included the core testing methodology, was developed by John Tsantilis.

8. References

[1] http://www.businessofapps.com/data/app-revenues/

[2] Efthimios Alepis and Constantinos Patsakis, 2017: "Unravelling Security Issues of Runtime Permissions in Android" https://www.researchgate.net/publication/328515601_Unravelling_Security_Issues_of_Runtime_Permissions_in_Android

[3] http://gs.statcounter.com/os-market-share/mobile-tablet/worldwide/#monthly-201208-201810

[4] https://www.openhandsetalliance.com/

[5] https://clevertap.com/blog/understanding-android-permissions/

[6] https://developer.android.com/about/versions/marshmallow/android-6.0-changes

[7] https://developer.android.com/guide/topics/permissions/overview

[8] https://developer.android.com/guide/topics/manifest/permission-element

[9] http://cloak-and-dagger.org/

[10] https://www.news.gatech.edu/2017/05/21/combination-features-produces-new-android-vulnerability

[11] TUNCAY, G.S., DEMETRIOU, S., GANJU, K., GUNTER, C.A.: RESOLVING THE PREDICAMENT OF ANDROID CUSTOM PERMISSIONS. IN: ISOC NETWORK AND DISTRIBUTED SYSTEMS SECURITY SYMPOSIUM (NDSS) (2018) http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018_08-4_Tuncay_paper.pdf

[12] Knock-Knock: The unbearable lightness of Android Notifications https://arxiv.org/pdf/1801.08225.pdf

[13] FRANCOIS-NICOLA DEMERS AND JACQUES MALENFAN "REFLECTION_IN_LOGIC_FUNCTIONAL_AND_OBJECT-ORIENTED_PROGRAMMING_A_SHORT_COMPARATIVE_STUDY", PROC. OF THE IJCAI'95 WORKSHOP ON REFLECTION AND METALEVEL ARCHITECTURES AND THEIR APPLICATIONS IN AI. PP. 29–38. AUGUST 1995, https://www.researchgate.net/publication/2732177_Reflection_in_logic_functional_and_object-oriented_programming_a_Short_Comparative_Study

[14] BENJAMIN LIVSHITS, JOHN WHALEY, AND MONICA S. LAM "REFLECTION ANALYSIS FOR JAVA", https://suif.stanford.edu/papers/aplas05r.pdf

[15] DANIEL G. BOBROW XEROX PARC, RICHARD P. GABRIEL LUCID, INC., JON L WHITE LUCID, INC., "CLOS IN CONTEXT:THE SHAPE OF THE DESIGN SPACE", MAY 3, 2004, https://extravagaria.com/Files/clos-book.pdf

[16] Smith, B.C. Reflection and Semantics in Lisp. Proceedings of ACM Symposium on Principles of Programing Languages (May 1984). http://research.cs.queensu.ca/~cordy/cisc860/Biblio/hurd/cs/smith84.pdf

[17] Reflection for the Masses. Charlotte Herzeel, Pascal Costanza, and Theo D'Hondt. Vrije Universiteit Brussel http://www.p-cos.net/documents/s32008.pdf

[18] https://docs.oracle.com/javase/tutorial/reflect/index.html

[19] https://docs.oracle.com/javase/8/docs/technotes/guides/reflection/index.html#a0v1

[20] https://developer.android.com/guide/platform/

[21] https://www.statista.com/chart/15561/smartphone-sales-by-os/

[22] Proc. of the First Workshop on Reflection andMetalevel Architectures in Object-Orient ed Program-ming, OOPSLA/ECOOP'90, Oct. 1990.

[23] https://www.linux.com/learn/overview-linux-kernel-security-features

[24] https://developer.android.com/reference/android/content/Context

[25] Using fuzzing to detect security vulnerabilities, INFIGO-TD-01-04-2006, 25-04-2006: https://www.infigo.hr/files/INFIGO-TD-2006-04-01-Fuzzing-eng.pdf

[26] Violating assumptions with fuzzing, P. Oehlert, IEEE Security & Privacy (Volume: 3 , Issue: 2 , March-April 2005) https://ieeexplore.ieee.org/document/1423963

[27] Research On The Generation Method Of Test Cases In Fuzzing https://www.atlantis-press.com/proceedings/icmii-15/25844363

[28] Miller B P, Fredriksen L, So B. An empirical study of the reliability of UNIX utilities [J].

Commun. ACM, 1990 ftp://ftp.cs.wisc.edu/paradyn/technical_papers/fuzz.pdf

[29] https://en.wikipedia.org/wiki/Fuzzing