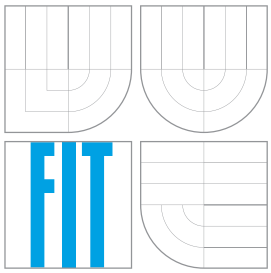
BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ



FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

DYNAMIC SECURITY POLICY ENFORCEMENT ON ANDROID

DYNAMICKÁ ÚPRAVA BEZPEČNOSTNÍ POLITIKY NA PLATFORMĚ ANDROID

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR
AUTOR PRÁCE

Bc. MATÚŠ VANČO

SUPERVISOR
VEDOUCÍ PRÁCE

Ing. LUKÁŠ ARON

BRNO 2016

Abstract

This work proposes the system for dynamic enforcement of access rights on Android. Each suspicious application can be repackaged by this system, so that the access to selected private data is restricted for the outer world. The system intercepts the system calls using Aurasium framework and adds an innovative approach of tracking the information flows from the privacy-sensitive sources using tainting mechanism without need of administrator rights. There has been designed file-level and data-level taint propagation and policy enforcement based on Android binder.

Abstrakt

Tato práce navrhuje systém pro dynamické vynucování přístupových práv pro platformu Android. Každá podezřelá aplikace může být zabezpečena tímto systémem tak, že je znemožněn únik citlivých dat mimo zařízení. Systém zachycuje systémová volání s použitím Aurasium framework, a přidává nový přístup sledování informačních toků z citlivých zdrojů s použitím systému značkování tak, aby nepotřeboval administrátorská práva. V práci bylo navrženo sledování dat na úrovni souborů a obsahu souborů, a vynucování bezpečnostní politiky vycházející z technologie Android binder.

Keywords

private data, Aurasium framework, operating system, system call, binder driver, Android security, policy enforcement, security policy

Klíčová slova

soukromá data, Aurasium framework, operační systém, systémové volání, binder driver, bezpečnost Androidu, vynucování zásad, bezpečnostní politika

Reference

VANČO, Matúš. *Dynamic Security Policy Enforcement on Android.* Brno, 2016. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Aron Lukáš.

Dynamic Security Policy Enforcement on Android

Declaration

I declare, that this diploma thesis is my independent work under the guidance of Ing. Lukáš Aron. I did not use any other sources, figures or resources than the ones stated in the bibliography.

.....
Matúš Vančo
May 24, 2016

Acknowledgements

I would like to thank my supervisor Ing. Lukáš Aron for his support, motivation and good mood. I would also like to thank Prof. Tomáš Vojnar and Dr. Petr Peringer for provided further information.

© Matúš Vančo, 2016.
This thesis was created as a school work at the Brno University of Technology, Faculty of Information Technology. The thesis is protected by copyright law and its use without author's explicit consent is illegal, except for cases defined by law.

Contents

1 Introduction **6**

2 Android Security **8**
- 2.1 Operating Systems Concepts . . . 8
- 2.2 Android Architecture . . . 11
- 2.3 Security Policy Enforcement . . . 13
- 2.4 Analysis of Security Issues . . . 17
- 2.5 Security Frameworks Comparison . . . 19
- 2.6 Jeon et al.'s Tools . . . 22

3 Aurasium Framework **24**
- 3.1 Principle of Mediation . . . 25
- 3.2 Static Analysis . . . 26

4 Analysis and Design **29**
- 4.1 Android Binder . . . 30
- 4.2 Command Line Tools . . . 33
- 4.3 System Design . . . 35

5 Software Development **38**
- 5.1 Integration . . . 38
- 5.2 Experiments . . . 39
- 5.3 Implementation . . . 41
- 5.4 Configuration . . . 45

6 Verification **47**
- 6.1 Testing . . . 47
- 6.2 Evaluation . . . 51

7 Conclusion **53**

Literature **55**

Appendices **59**
- List of Appendices . . . 60

A Pseudocodes **61**

B Application Screenshots **63**

C DVD Content 65

List of Figures

2.1 Interaction Diagram of Linux Kernel . . . 10
2.2 Protection Rings in Operating Systems[1] . . . 11
2.3 Android Components . . . 13
2.4 Forms of Component Interaction [37] . . . 13
2.5 Permission Label Model . . . 15
2.6 Taxonomy of Permission Escalation Attack [16] . . . 17
2.7 Relationship among Issues in Android Security [16] . . . 18
2.8 Countermeasures according to Addressed Issue . . . 19
2.9 Jeon et al.'s Tools Architecture [26] . . . 23

3.1 Aurasium Architecture [44] . . . 24
3.2 Mapping into Process's Address Space . . . 26
3.3 Aurasium Static Analysis . . . 27

4.1 Semantics of `binder_write_read` Structure[2] . . . 31
4.2 Binder Architecture[3] . . . 32
4.3 Smali Principle . . . 34
4.4 Tainting Principle [12] . . . 35
4.5 Design of Architecture . . . 36

5.1 Analysis of Interception . . . 40
5.2 Overview of the Tainting System . . . 42
5.3 Principle of Content-based Scanning . . . 44
5.4 Interacting with a Content Provider [42] . . . 45
5.5 The Syntax of Configuration File . . . 46

6.1 Peformed Tests According to Hierarchy Level . . . 47
6.2 Size Increase After Repackaging in Original Aurasium [44] . . . 50

A.1 Pseudocode of `binder_write_read` Structure . . . 61
A.2 Pseudocode of `binder_transaction_data` Structure . . . 61
A.3 Pseudocode of Data Structures for Tainting . . . 62

B.1 Main Screen of Configuration Activity . . . 63
B.2 Selection of Private Files . . . 63
B.3 Settings in Configuration Activity . . . 63
B.4 Information Screen in Configuration Activity . . . 63
B.5 Sample of Sharing Possibilities . . . 64
B.6 Sample of Confirmation Dialog . . . 64
B.7 Sample of Restriction Using Communication Mediation Method . . . 64

B.8 Sample of Restriction Using File Protection Method 64

List of Tables

2.1 Standard Activity Actions [1] 14
2.2 Comparison of Selected Countermeasures 20
2.3 Findings of Yukse's et al. Assessment [45] 22

4.1 Intercepted System Calls in Aurasium 30
4.2 Binder Driver Commands 32
4.3 Android Debug Bridge Commands [1] 34

5.1 Function Prototypes of System Calls 39
5.2 Captured Call Transactions 41
5.3 Captured Return Transactions 41
5.4 Summary of Log Files 42

6.1 Testing of Final Restriction in OI File Manager Application 49
6.2 Repackaging Evaluation Results [44] 49
6.3 Startup Time Slowdown on Repackaged Applications 49
6.4 Performance of Sharing Actions with Restriction 50
6.5 Testing of Tainting Performance 51
6.6 Time of File Opening During Increased Threat Level 51

Chapter 1

Introduction

Android's fast growth of popularity has a lot of causes and consequences. Growing number of applications, increasing number of devices and growing level of integration have been interfered and influenced each other, implying increasing volume of the *private data*. People have put their trust in their devices and become more dependent on mobile technologies, using them for socialization, trading or entertainment. However, the private data is being used for the profit still more often during globalization, because it is the base for the knowledge-based business and targeted advertising. Even Google's free Android generates the significant part of its revenue just this way [36].

Since this asset is seized by many groups of people using illegal ways, Android has become the most assaulted mobile operating system facing the wave of malware, which is even more capable and stealthy, and can even establish a permanent presence on the device [44]. In order to address these challenges, Android includes permission model that protects access to sensitive resources. However, since permissions are overly broad and misunderstood, applications are provided with more access than they truly require. In particular, they are granted statically during install-time and thus do not correspond to the actual use at the time. This implicates big vulnerability even if a certain application is not intended to misuse the private data because it can be exploited.

Based on this insufficient built-in Android security and his later refinements, plenty of third-party frameworks seek to supplement overall security. The current state of the art comprises several effective countermeasures to issues like the coarse granularity of permissions, over-claim of permissions and permission escalation attack. However, most of these solutions are rather too complex and less straightforward. They replace the whole Android permission model, or try to track the information flow on the level of the operating system which requires the rooted device. In contrast, there is Aurasium framework, which automatically repackage and harden chosen applications, interposing the sandboxing code in the applications themselves. Nevertheless, it is robust enough to interpose almost all types of interactions between the application and the operating system.

The aim of this work is to develop the system, which utilizes the Aurasium framework and restricts the access of private data outside the device through the selected applications. In contrast to original Aurasium framework, this work focuses more on the real asset, the private data, and especially on the high usability and easy deployment. The solution is focused on tracking the information flows from the privacy-sensitive sources to the system sink where they aim to leave the system.

The first chapter, *Android Security*, deals with the two independent theoretical basis – the state of art in Android and related Linux OS security as well as the comparison of

third-party refinements. In the chapter *Aurasium Framework*, outputs of the Aurasium framework research and code analysis are summarized. The next chapter, *Analysis and Design*, describes the selected theoretical building blocks, decisions and design conducted prior to the implementation. After that, the work performed during the project is being overviewed in the chapter *Software Development*. The last chapter named *Verification* is concerned with the testing methodology and evaluation of the overall project in detail.

Chapter 2

Android Security

Android security has been built upon fundamental security concepts of operating systems themselves. Since the security in general rests on the *confidentiality*, *integrity* and *availability* as well-known concepts in IT security discipline, the OS security is also intended to lower the probability of security incidents, especially their impact on the valuable assets it manages [7, 24]. The purpose of an operating system is to create the interlayer between computing hardware on one side and the users and their application-layer programs on the other side. It manages hardware resources like processor, memory and hardware peripherals and creates an environment for users and application programs [40]. Therefore, OS is essential software for overall security and privacy, since the security of user programs cannot be better than the software layer that it utilizes underneath. In another case, even secure and well-programmed applications can interrupt integrity and stability of the system. Moreover, operating systems have been still improving and additional complexity creates more vulnerabilities which can be exploited. Several users want to work in parallel on multiple tasks, use various peripherals at the same time and still experience secure as well as responsive system. To adapt to these requirements, a lot of security technologies have and must have been integrated into modern systems including preemptive process planning, advanced access control of memory resources and are required to recognize a variety of peripherals on the way. Android, in addition, utilizes mandatory access control and permission label model on the top of OS kernel and introduces several refinements for secure programming. Based on insufficient built-in Android security, there are also plenty of third-party frameworks which are intended to supplement overall security.

This chapter is divided into two parts. In the first part, built-in Android security is reviewed on the level of operating system, on the level of Android middleware architecture and on the level of usage from developer's perspective. This is included in sections *Operating Systems Concepts*, *Android Architecture* and *Security Policy Enforcement* in that order. The second part of this chapter is a literature review of security issues regarding built-in security and their respective existing third-party countermeasures. There is also performed analysis and comparison of this countermeasures. This is an objective of the sections *Analysis of Security Issues* and *Security Frameworks Comparison*.

2.1 Operating Systems Concepts

In terms of operating systems, the main aim of security is to ensure that the processes in an OS are protected from one another's activities. To provide such protection, there are

various mechanisms to ensure that only processes that have gained proper authorization from OS can operate on the resources of a system. This is performed with basic *capability-based* security concept – grant the lowest possible required permissions to every application to perform its functionality without restricting other applications installed in the system. This is known as the *principle of least privilege* (PLP). In security, there is a *subject*, requesting for an access while using granted permissions and an *object*, data resource which can define access permissions for the protection. Conceptually, if the subject requires access to shared object, it must check if it has granted proper permission or in other words if it is *authorized*. Otherwise, OS is blocking access. Furthermore, at any time, a subject should be able to access only those resources that are currently required to complete its task. This requirement is commonly referred to as the *need-to-know principle* [40]. Checking has been usually performed via *Access Control List* (ACL) where the security policy is defined. This mechanism is called Mandatory Access Control (MAC), because security policy is invariably stored and mandatory enforced by the system. In contrast, *Discretionary Access Control* (DAC) is less strict, and competence for defining a security policy is delegated to the user and his discretion. The main information sources for this section are books *Operation System Concepts* [40] and *Modern Operating Systems* [42].

Regarding security countermeasures in OS process management, processes in the modern operating systems have been administrated with *preemptive process planning*, concept utilizing hardware mechanism – interruption. Preemptive planning ensures better stability and security and is present on the majority of today's operating systems. While in cooperative or non-preemptive multitasking, applications are equal and have the right to make their own decision about needed processor time, preemptive multitasking uses the principle of least privilege and the operating system is in charge to utilize time-division multiplexing, assigning the small amount of time for each application. This prevents deadlock and selfish misuse of resources by one application. As the computer cannot distinguish between system code and application code by default, there must be cooperation with hardware. The processor has been scheduled to interrupt application execution after time quantum is expired and returns its operation to the planned block of code according to *Interrupt Vector Table* containing addresses of following instructions after interrupting.

The Linux is considered as the main referential operating system nowadays since it is the most used and taught system in the academic sphere. Linux kernel is currently part of the uncountable desktop Linux distributions as well as the most mobile devices including Android platform and other mobile operating systems such as Firefox OS [29], Chrome OS [22] and Tizen [28]. From the security perspective, Linux utilizes a lot of security concepts which are directly used by Android operating system and its security infrastructure.

A Linux-based system is a modular Unix-like operating system, deriving much of its basic design from principles of Unix. It uses a Linux kernel, which handles access to the file systems, peripherals and handles process control. It runs in protected kernel-mode and is not accessible from application-layer programs running in the user-mode [fig. 2.1]. This architectural approach to protection is technically called *hierarchical protection domains* or *protection rings* and is contrasted to that of capability-based security approach. The illustration is shown in figure 2.2 with the most common use of protection levels in italics. The protection domain is a set of access rights, each of which is an ordered pair `(<Object Name>, {<Access Right>, ...})`. The first element of the pair is the full identification of exact physical hardware device (disk, printer, CPU, ...) or software object (file, program, semaphore, ...). The second element is a set containing access rights which are dependent on the particular object. The kernel provides an application interface to upper layers – kernel

services. There is a possibility to request service execution by performing *system call*. However, system calls are usually not invoked directly. Instead, most system calls have corresponding C library wrapper functions which perform the required steps. Example system calls are shown in the next figure.

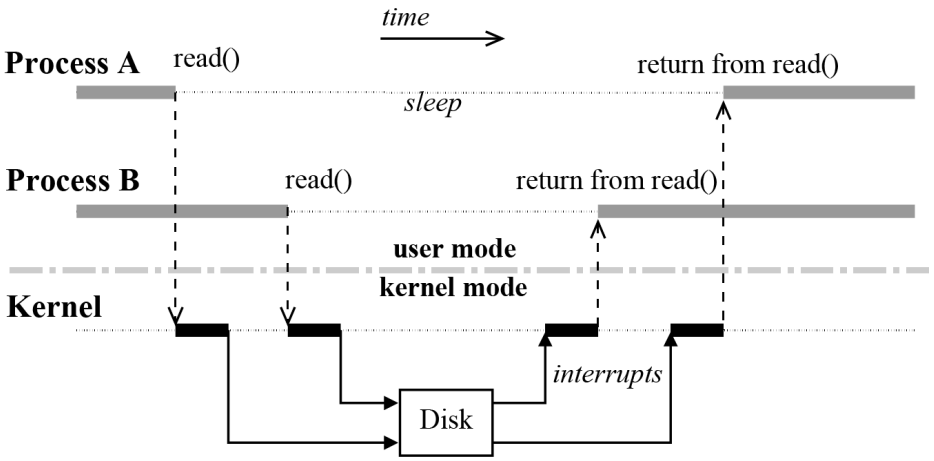


Figure 2.1: Interaction Diagram of Linux Kernel[1]

Returning to process management in OS, there is other very important concept especially in terms of Android security. Processes executing concurrently in the operating system may be cooperating processes. They can affect or be affected by other processes and share their data with any other processes. Therefore, there is needed *interprocess communication* (IPC) mechanism that will allow this kind of exchanging data. There are two fundamental models of interprocess communication – *shared memory* and *message passing*. In the shared-memory model, a region of memory that is shared by cooperating process is established, while in the message-passing model, communication takes place by means of messages exchanged between the cooperating processes. In client-server systems, there are three other strategies for communication – sockets, remote procedure calls (RPC), and pipes. The most important strategy in the context of this work, is RPC. The RPC was designed as a way to abstract procedure call mechanism for usage between systems with network connections. In many respects, it is similar to the IPC mechanism and it is usually built on top of such a system. However, in this case, message-based communication scheme is used in order to deal with an environment in which the processes are executing on separate systems.

The last selected security concept in OS, which will be mentioned, is related to the storage management responsibility of OS. The operating system provides a uniform logical view of stored information, abstracting from the physical properties of its storage devices and defines a logical storage unit, the *file*. Since files and contained information are assets, as described in the previous section, it must be kept safe from improper access ensuring confidentiality as well as availability. The most common approach to this protection problem is to make access dependent on the identity of the user as different users may need different types of access to a file or directory. This identity-dependent access is implemented by associating ACL with each file and directory. In the ACL, there are pairs with specified user names and their corresponding type of allowed access. The main problem with this type of access list is its length. If there is a need to allow everyone to read a file, there must

[1]Operating Systems course at BUT FIT

be listed all users with read access. Therefore, UNIX system recognizes three classifications of users in connection with each file – an *owner*, the user who created the file, a *group*, a set of users who are sharing the file and need similar access and *other*, all other users in the system.

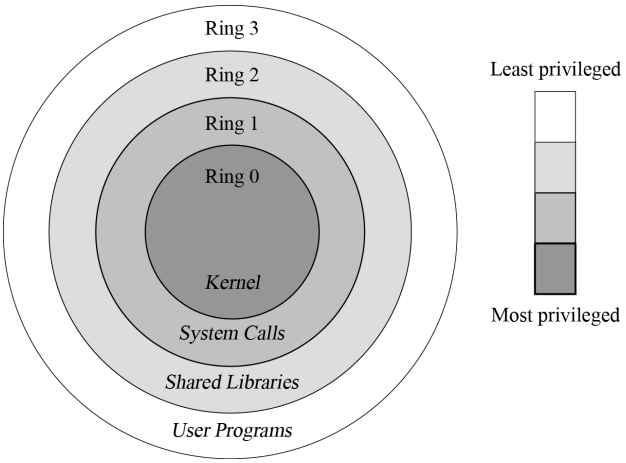


Figure 2.2: Protection Rings in Operating Systems[2]

2.2 Android Architecture

Android OS [1] is the mobile operating system built upon Linux OS. OS is developed by the Open Handset Alliance [33] (OHA) with the biggest supporter Google Inc. Open Handset Alliance was established in 2007 when the Android SDK has already been publicly available. OHA associates dozens of the biggest companies in software/hardware and telecommunications industry including mobile operators, software companies, handset makers and chip makers [11]. It is currently the most popular mobile operating system with billions of devices shipped. In addition, it is the main OS platform supporting the *Internet of Things* (IoT) or *Internet of Everything* (IoE) environment, which has evolved from the convergence of wireless technologies, *micro-electromechanical systems* (MEMS) and the Internet. This is caused by deployment of Android on devices and things like tablets, phablets, smart wearables (watches, fitness bands, glasses, virtual-reality devices, clothes, ...), health devices (personal weight scales, thermometers, pedometers, blood pressure monitors), smart toys, cars, kitchen and household appliances (fridge, freezer, washing machine), smart home devices (smart bulbs, security systems, monitoring systems, security cameras), smart TVs and other intelligent sensor devices. This also implies the main selling point of Android – capability to easily extend online services to mobile devices. This extremely extensive environment has been formed very quickly and Android system must be still evolving and must ensure enhanced security.

Android security model has been built upon Linux security mechanisms listed in previous sections. The most important security mechanism is sandboxing. Sandboxing is a mechanism to isolate applications from each other and from system resources. Application isolation is done by means of assigning a unique user identifier (UID) to each application, while the underlying Linux kernel enforces discretionary access control to resources (files

[2]Inspired by the book Operating system concepts [40]

and devices) by user ownership [9]. Android uses reliable separation of processes in Linux and utilizes it to separate user applications. Each application is primarily written in Java programming language and is running on the similar virtual machine as Java Virtual Machine (JVM). Dalvik Virtual Machine [1] (DVM) has been developed due to adaptation for systems with constrained resources like memory or processor speed. Each application runs with its own DVM as stand-alone Linux process and under unique Linux user, so application data is protected against access by other applications. Communication between applications is realized by using Linux interprocess communication mechanism as has been described before. From the perspective of the developer, communication between applications is the same as the communication between application components. Therefore, it is important to know what the Android applications consist of. This literature review is mainly security-oriented, but some additional details can be found in the earlier thesis about Android [43].

Android SDK prepares four *application components*, java classes, which should be used by developers during building their applications. Namely, it is Activity, Service, Broadcast Receiver and Content Provider. These components are shown in figure 2.3. Each Android application can use none or several components of each component type. There is one possible situation with one foreground application and one background application illustrated in the figure, which can be started, but the user cannot see it and is not interacting with it at the moment. Component type *Activity* is used for defining the user interface (UI) and interaction with the user. There must be only one foreground Activity in the system at the same time. Activity can already include all application logic, in the actual class, but a better design is to include all application computing and data processing in a separate class, java package or even in another Android application package. Another application component, *Broadcast Receiver* has an intent to work like helm or steering wheel for other applications. It can directly affect and control application processing and behaviour, but it does not create UI. For instance, application for photo-editing can define Broadcast Receiver in order to start itself when another application wants to edit some picture, but cannot do it alone. The *Service* is a component for data processing in the background without UI, even if the application is not in the foreground. Service usually communicates with a Web server and is performing time-consuming operations such as synchronizing application data to cloud service. By default, all application components are performed in single thread – *UI thread* or *main thread*. Therefore, it is necessary to create new thread explicitly in this component so as to prevent the UI jams. The last component, *Content Provider*, has been built upon SQL database, which it wraps and included data is made available and addressed via an authority string embedded in a special content URI of the form `content://<authority>/<table>/[<id>]` [15].

These application components interact with each other and use intent mechanism. In Android, the Intent represents data which need to be processed with a meaning or a definition how it should be processed or what type of data it is. Example of intent is a picture with “VIEW” action string. Action string defines action how the data should be processed. Intent can be also imagined as a message which is sent from one component into the another and it contains information with which intent the data should be processed in the receiver. The list of standard Activity actions is depicted in table 2.1.

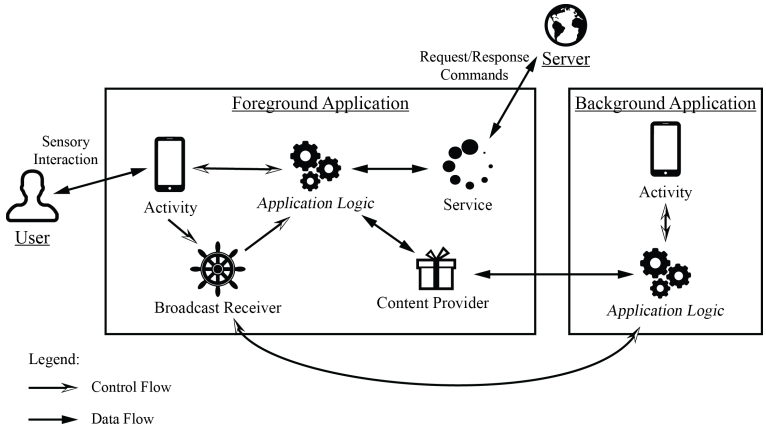


Figure 2.3: Android Components

2.3 Security Policy Enforcement

Communication between components is maintained using *Reference Monitor* (RM), which is part of Android OS Middleware. It ensures communication between applications separated on the OS level and permission label mediation and mandatory access control (MAC). ACL in Android OS is statically defined and user-authorized during installation and does not provide an ability to further modify or grant the subset of manifested requested permissions as it is possible in the Windows Phone OS. Reference Monitor, illustrated in the figure 2.5, utilizes Inter-component Communication (ICC) similar to IPC in Unix-based systems.

However, in this case, ICC is performed via shared-memory IPC [sec. 2.1] and uses special file `/dev/binder` representing special device node in Linux. Because the file must be readable and writable for everyone due to proper operation, the Linux system has no way of mediating it with its security mechanisms [15]. ICC is then performed via an input or output control command on this special device node. Android utilizes this communication in several different forms according to involved components which is shown in figure 2.4.

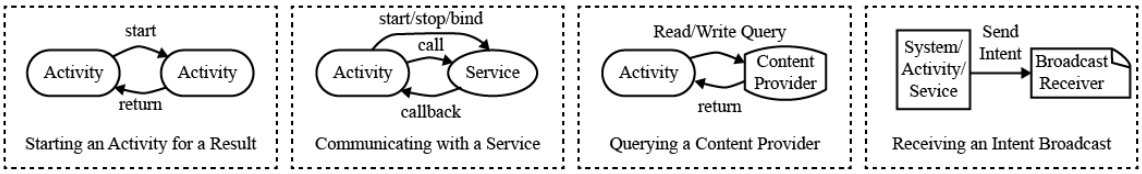


Figure 2.4: Forms of Component Interaction [37]

Security enforcement using MAC uses two types of permisions – *granted permissions* (used or requested permissions) and *required permissions* (access permissions) [9]. Granted permissions can be manifested during installation and are inherited by all of the application's components. On the other hand, required permissions are usually created by the developer to protect their important components. Required permissions are always assigned to application components separately. In addition to this division, Android defines four protection levels for each permission according to significance and actual meaning [1]:

- **Normal** – Low-risk permissions causing no harm to users. Permissions are automatically granted when application declares request.

Action	Description
ACTION_MAIN	Start as a main entry point, does not expect to receive data.
ACTION_VIEW	Display the data to the user.
ACTION_ATTACH_DATA	Indicate that some piece of data should be attached
ACTION_EDIT	Provide explicit editable access to the given data.
ACTION_PICK	Pick an item from the data, returning what was selected.
ACTION_CHOOSER	Display an activity chooser.
ACTION_GET_CONTENT	Allow the user to select a particular kind of data and,return it.
ACTION_DIAL	Dial a number as specified by the data.
ACTION_CALL	Perform a call to someone specified by the data.
ACTION_SEND	Deliver some data to someone else.
ACTION_SENDTO	Send a message to someone specified by the data.
ACTION_ANSWER	Handle an incoming phone call.
ACTION_INSERT	Insert an empty item into the given container.
ACTION_DELETE	Delete the given data from its container.
ACTION_RUN	Run the data, whatever that means.
ACTION_SYNC	Perform a data synchronization.
ACTION_PICK_ACTIVITY	Pick an activity given an intent, returning the class,selected.
ACTION_SEARCH	Perform a search.
ACTION_WEB_SEARCH	Perform a web search.
ACTION_FACTORY_TEST	Main entry point for factory tests.

Table 2.1: Standard Activity Actions [1]

- **Dangerous** – High-risk permissions which allows applications to access potential harmful API calls. User is warned during installation time and must approve access to these resources.
- **Signature** – A possible high-risk permission for some application. Permissions are automatically granted when the requesting application is signed with the same certificate as the application, which defines the permission, is signed.
- **Signature-or-system** – A possible high-risk permission for Android system. Permissions are automatically granted when the application is in the same Android system image or is signed with the same certificate as the application which defines the permission is signed.

Android restricts application interaction to its API, which uses *permission label assignment model* [15]. Permission label is a simple text string representing granted or required permission mentioned above. According to the documentation for Android developers, there are currently 130 permissions defined in Android Developers page ranging from access to a camera (`CAMERA`), full access to the Internet (`INTERNET`), dialing a phone number (`PHONE_CALL`), and even disabling the phone function permanently (`BRICK`) [1]. In addition to Android defined permissions, application developers can also declare customized permissions so as to protect their own critical resources. Permission label in Android does not have exactly defined meaning and access to labels is not restricted [15]. This shortcoming causes the main problems in Android security. All permissions are declared in Android Application Package (APK) in the XML file called `AndroidManifest.xml`. Manifest file should include the whole security policy and serve as ACL mentioned before [sec. 2.1]. However, this is not entirely true, since Android designers have added several potentially confusing refinements as the system has evolved. Some refinements push policy into the application, others add delegation and that is why security analysts cannot discover an application's

policy simply by looking at it. APK file is merely a Java JAR archive containing except AndroidManifest.xml the application's code in the form of dex bytecode, compiled XML resources such as window layout and string constant tables, and other resources like images, sound and native libraries [44]. It also includes its own signature in a form identical to the standard Java JAR file signatures.

Detailed principle of PAM is illustrated in figure 2.5. Required permissions are assigned to system services. The service S_A has assigned permission P_1 and the service S_B permission label P_2. Application 2 and application 3 are in the same sandbox, because they are signed with the same key and they have been requested to share the same UID. Both applications have granted permissions P_1 and P_3. This means, that every component in every application in this sandbox has been allowed to access components with required permission P_1 or P_2. Service in application 2 and service in application 3 can, therefore, access the system service in system sandbox. Components without assigned required permission have been unprotected and are accessible by any other application component. However, the protected system service S_A cannot be accessed by application 1 and the system service S_B by application 2, because in their package any granted permission P_1, respectively P_2, is not manifested

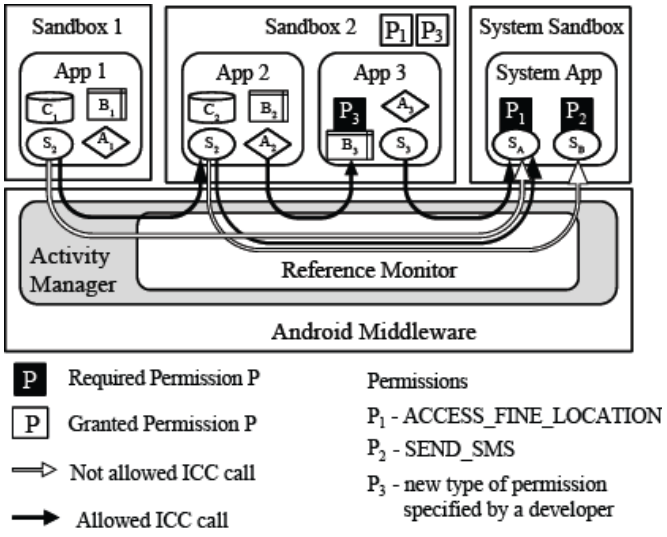


Figure 2.5: Permission Label Model[3]

Based on illustration for better understanding, a simple mathematical model can be proposed.

Definition 2.3.1. Let β be an overall security policy on Android. Then the β is a binary relation from a set of all installed applications A to infinite set of all available and later user-defined permissions P. It can be also defined as a function from the set of installed applications A to the power set of set of permissions P denoted as $\mathcal{P}(P)$.

$$\beta : A \to \mathcal{P}(P),\ \beta \subseteq A \times P,\ where\ A = \{A_1, A_2, ..., A_n\}, P = \{P_1, P_2, ..., P_n\} \tag{2.1}$$

Each application has a minimal set of permissions needed for its operation and actually

[3] Adapted from Bugiel et al.'s article [9]

requested by policy during install-time:

$$P_{min}, P_{act} \in \mathcal{P}(P) \tag{2.2}$$

$$P_{min} \subseteq P_{act} \tag{2.3}$$

Properly defined security policy for Android application:

$$P_{min} = P_{act} \; or \; P_{min} \doteq P_{act} \tag{2.4}$$

Statically defined permissions are too coarse-grained because combinations of permissions could be required by any third-party application for its legitimate functionality, and at the same time, these combinations of permissions could be harmful if they are exercised by any malicious application in a specific sequence [5]. Based on previously defined model, we can define a generic application runtime behaviour as a graph and specific snapshot behaviour using sequence of permissions similarly:

Definition 2.3.2. An application runtime behaviour graph is a directed acyclic graph (S:P) where S is a set of nodes representing states of an application, and P is set of edges representing different permissions exercised.

Definition 2.3.3. Snapshot behaviour is sequence of permissions defined as ordered multiset:

$$P_{Seq} = (P_{i_1}, P_{i_2}, ..., P_{i_n}), \; where \; \forall x \in \{1, 2, ..., n\} : i_x \in \{1, 2, ..., |P|\} \tag{2.5}$$

Permission sequence $(P_{i_1}, P_{i_2}, ..., P_{i_n})$ can be rewritten to be more illustrative as:

$$P_{i_1} \rightarrow P_{i_2} \rightarrow ... \rightarrow P_{i_n} \tag{2.6}$$

Definition 2.3.4. Let δ be a function that defines malicious behaviour. Then δ projects arbitrary permission sequence to decision set D = {M, B}, where M and B represents malign respectively benign sequence:

$$\delta : P_{Seq} \rightarrow M, B \tag{2.7}$$

For example if P_1, P_2 and P_3 represents `INTERNET`, `READ_CONTACTS` and `CALL_PHONE` permissions in that order, sample VOIP application could be considered to be malicious respectively benign followingly:

$$\delta(P_1 \rightarrow P_2 \rightarrow P_3) = B \tag{2.8}$$

$$\delta(P_2 \rightarrow P_1 \rightarrow P_3) = M \tag{2.9}$$

Regarding security refinements, Google developers, who designed Android, incorporated several refinements to the basic security model. The main aim was to improve programmers convenience and partially to include necessary hooks. Such refinements are permission protection levels which have already been described. The first security refinement is related to object-oriented paradigm and encapsulation principle. In Android, the component can be marked as *private* which implies, that all external components are forbidden to access it even if there is no required permission assigned [15]. This mechanism should be used on the all internal components, because component without assigned required permission is always

insecure and *implicitly open*. The next security refinement is the one of the necessity. It is concerned to broadcast intents as they are vulnerable to permission escalation attack. *Permission escalation attack* (PEA) can be categorized into *Confused Deputy Attack* (CDA) and *Collusion Attack* (CA) [fig. 2.6]. CDA exploits unprotected interfaces of privileged benign applications. For example, even if the *application 1* in figure 2.5 cannot access system service S_A directly, because it has not been manifested and granted the permission P_1, application can access the *application 2* or *3* and misuse its granted permissions to get where it has wanted. Another attack, CA, uses the principle of generating a joint set of permissions from multiple malign applications. For example, if the *application 1* was granted permission P_2, both the *application 1* and *2* could collusively cooperate with the aim to misuse the united permissions. CA requires the cooperation of multiple malign applications using direct or indirect communication via third application or component. Indirect CA can be performed not only by using *overt channel*, such as buffers, files or I/O devices, but also *covert channels* including file locks, change of screen state or type of intents. One of the covert channels can be also broadcast intent, since every application is capable of reading every intent without restriction. For this vulnerability to be addressed, Android has introduced *broadcast intent permissions*, which works in reverse as common granted permissions. These permissions provide confidentiality of an author or creator of intent. The *Content Provider Permission* refinement introduces finer-grain permission protection enabling assignment of two permissions to one Content Provider component – READ and WRITE. These permissions are automatically granted to requesting applications, which are then allowed to perform enabled operation (none, read, write or read and write). *Service hooks* enables finer granularity of RPC calls in Service component type and *Protected APIs* forces security to resources, which are implemented directly without the use of API. This is related to system services like a camera and other resources implemented directly upon OS itself. [15]

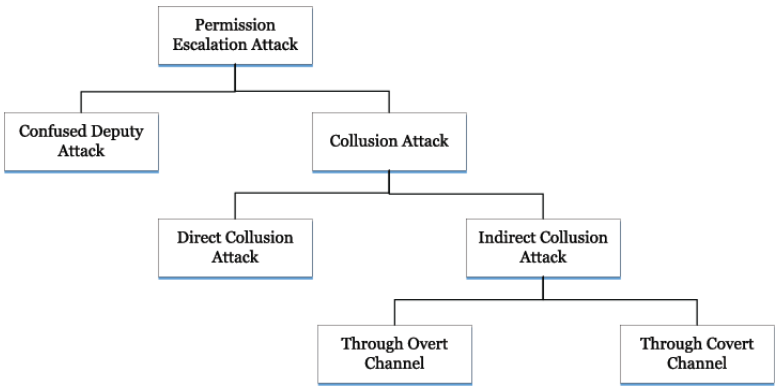


Figure 2.6: Taxonomy of Permission Escalation Attack [16]

2.4 Analysis of Security Issues

Mass deployment of Android OS on the various devices [sec. 2.2] and increased popularity have caused that a lot of developers, driven by popularity, reputation and proceeds, have engaged developer community [11] which has implied a rapid increase in the number of applications available in the Google Play Store. Simultaneously, the number of malign applications has been increased. In the present, even regular user is capable of download-

ing rooting application from an unofficial source and performing the permission escalation attack disabling all security mechanism which were presented. Fortunately, there are some promising third-party projects and frameworks, which are focusing on enhancing Android security and preventing privacy leakage. The main information source for this chapter is Fang et al.'s article [16].

From Fang et al.'s security analysis, some threats and vulnerabilities which could lead to risk have been identified. These issues are summarized in the following figure [fig. 2.7]. The issues are divided into two categories – *direct issues* and *indirect issues*. *Direct issues* are threats or vulnerabilities with a direct potential to lead to financial losses or to leak the private user information. *Indirect issues* addresses system properties, people's characteristics or other circumstances which simplify misuse of the security model in Android and thus, causes problems indirectly.

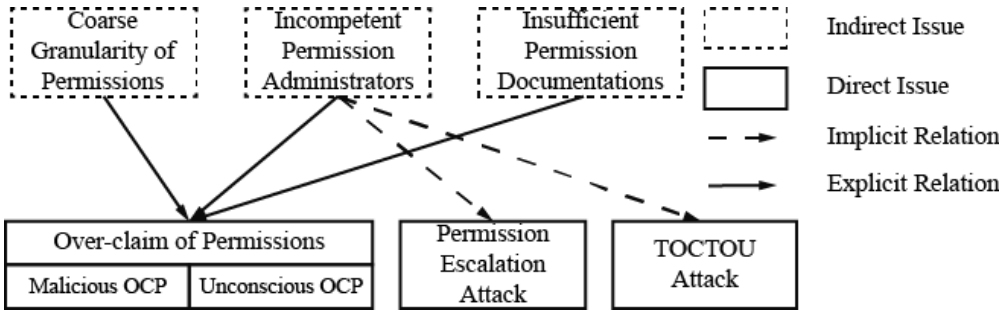


Figure 2.7: Relationship among Issues in Android Security [16]

Regarding *Coarse Granularity of Permissions* (CGP), Android defines permissions like `INTERNET`, `READ_PHONE_STATE` and `WRITE_SETTINGS` which provide an arbitrary accesses to certain resources without further specification. For example, a lot of applications use the `INTERNET` permission only for making HTTP(S) requests to specific domains or to support Google AdSense. However, permission administrator must also provide access to the other, possibly malicious, sites. Nevertheless, Android defines up to 300 permissions. This high number, together with insufficient knowledge or lack of interest, is the main factor of other indirect issue – *Incompetent Permission Administrators* (IPA). Permission administrators are usually developers and they are interested in the working application in any possible way, disregarding what is at risk for end-users. The last indirect issue, *Insufficient Permission Documentation* (IPD), can be also based on a large number of permissions as well as insufficient and imprecise documentation. Moreover, six errors are already identified in the documentation as well as it lacks dual interpretation for users, of which the technical description is often too complex and abstruse. All three mentioned indirect issues illustrated in the figure explicitly lead to direct issue OCP.

Over-claim of Permissions (OCP) addresses the situation when the application requests more privileges than it actually needs which can be misused and already identified as malicious behaviour. This issue directly breaks the principle of the least privilege and can be caused also by improper modular design where developers request for permissions which should be requested by deputy applications. OCP can be induced maliciously or unconsciously. Another direct issue, *Permission Escalation Attack* (PEA), is the main issue addressed by a lot of third-party countermeasures and has been described in the previous literature review. The last direct issue, the *TOCTOU* (Time of Check to Time of Use) *Attack*, is vulnerability of system that disables modification of security policy during the

time between checking of a condition (usually security credential) and use of the results of that check. This time can be dilated by an attacker or the attacker can directly skip the checking of a condition and use already granted authorization when it is the most suitable for him. TOCTOU attack is possible due to two factors – no naming rule is applied to a new permission in Android OS, and all permissions in Android are simple strings enabling any two permissions with the same name to be treated as equivalent even if they belong to unrelated applications. PEA and TOCTOU attack can be caused by indirect issue *Incompetent Permission Administrators* indirectly or implicitly, because administrators do not perform actual attack themselves.

2.5 Security Frameworks Comparison

Implementations of countermeasures to problems have been divided into several categories according to respective addressed issues. Compared implementations address the three of six mentioned issues – *Coarse Granularity of Permissions* (CGP), *Over-claim of Permissions* (OCP) and *Permission Escalation Attack* (PEA). These categories have been further divided to provide more detailed overview. Resulting categories and their overlapping have been illustrated in figure 2.8.

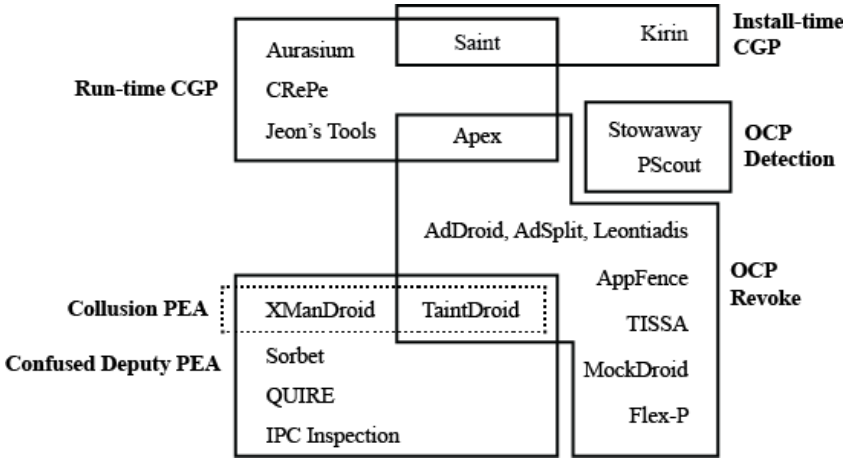


Figure 2.8: Countermeasures according to Addressed Issue

Enforcement can be performed during install-time or during run-time. Install-time enforcement means that applications that do not pass the established policy are prevented from being installed. On the other hand, run-time enforcement enables installation itself, but dynamically prohibits access to protected system resources or other application components. In that case, some applications exit unexpectedly with an exception, some can continue their activity with false data provided, some can prohibit starting an application in certain context. Also, information sources for establishing the policy rules are different. Some applications use policies according to user wishes, some analyze all existing permissions in the system or content of intents and creates rules accordingly to prevent various attacks, some are defined statically. In addition, the each solution has been working on the different layer of the Android OS [fig. 2.5] and only minority of them does not require administrator privileges to the system. Also the information source for the decision making about policy enforcement differs. It can be an author of application, a user of application or

there are used existing security policies to decide whether to allow or block certain action. Overview of some chosen countermeasures is in table 2.2.

	Est.	Prohibition Behaviour	Policy Source	Working Layer
Kirin [13, 14]	2009	Not Installed Application	Statically defined	Middleware
Saint [32]	2009	Not Installed Application	Applications	Middleware
Aurasium [44]	2012	Shown Dialog	End-users	Application
CRePe [5]	2011	Not Started Application	End-users	Middleware
Jeon's Tools [26]	2012	Exception Thrown	Applications	Application
Apex [31]	2010	Exception Thrown	End-users	N/A
AppFence [25]	2011	Transparent	End-users	Middleware
TISSA [46]	2011	Transparent	End-users	Middleware
MockDroid [6]	2011	Transparent	End-users	Middleware
Flex-P [30]	2012	Exception Thrown	End-users	Middleware
XManDroid [9]	2011	Exception Thrown	Content of Intents	Middleware & Kernel
TaintDroid [12]	2014	Exception Thrown	Applications	Middleware

Table 2.2: Comparison of Selected Countermeasures

Implementations in the CGP category have the aim to modify built-in policy enforcement using finer grains like IP address ranges, context information (location, time, system settings), specific resources (files), specific intents (according to action strings), application signatures, application versions or quantitative constraints like number of times a permission can be used. This category is further divided into Install-time CGP (IT-CGP) and Run-time CGP (RT-CGP) as it is illustrated in figure 2.8. The representative of the first category IT-CGP is *Kirin* [13, 14]. *Kirin* enforces install-time policy according to set of predefined security rules based on permission combinations and permission and action string combinations. *Saint* [32] is advanced solution with as install-time policy as a run-time policy enforcement. *Saint* uses *AppPolicy provider* which dynamically analyzes and identifies the appropriate policies according to transient state of phone and context, such as location, time, Bluetooth connection or connected devices. *Aurasium* [44] is together with the *Jeon et al.'s Tools* [26] including *RefineDroid*, *Mr. Hide* and *Dr. Android* the only solution working on the application layer without need of elevated privileges in the Android OS. However, this is redeemed by worse usability because every application must be repackaged and reinstalled in order to work with this system. *Aurasium*'s policies are user-defined. System uses blacklisting of IP addresses, protects against access to premium numbers and various system resources. *Aurasium* run-time monitors, attached to the application, execute in the same process as the application and hence are potentially subjects to circumvention. On the other hand, *Jeon et al.'s Tools* use taxonomy and systic analysis tool (*RefineDroid*), that infers the fine-grained permission usage. Applications are repacked by using the *Dr. Android* (Dalvik Rewriter for Android) tool. *Jeon et al.'s Tools* suffer to performance slowdown, because required interprocess communication by *Mr. Hide* (Hide Interface to Droid Environment) tool is an expensive operation. However, *Mr. Hide* service runs in a separate process, which is a little more secure than monitors in *Aurasium*. The last tool in the CGP category is *CRePE* [5] which is focused on business sector and uses *PolicyProvider* similar to *Saint*'s *AppPolicy provider* to enforce policy based on context – location, time and temperature.

Countermeasures addressing the Over-claim of Permission issue are coping with two problems – detection and analysis of permissions in Android permissions (OCP Detection) and proposals of enhanced frameworks which allow users to revoke over-claimed permis-

sions (OCP Revoke). Indirectly, all Coarse Granularity of Permissions issues are also Over-clam of Permissions issues, however, OCP countermeasures are focused more on detection, vulnerabilities in advertising libraries, granting of subset of permissions and limiting the install-time Android's permissions to secure system and protect against private-data misuse, while the CGP solutions, in addition, provide enhanced granularity using contexts, signatures, IP filtering or e.g. quantitative constraints. There is one solution overlapping with the previous Coarse Granularity of Permissions category, *Apex* [31], introducing the conditional grant of permissions. *Apex* enables selectively grant requesting permissions defined in Android manifest (addressing OCP) using a simple interface provided by *Poly*. Dealing with the issues related to Over-claim of Permissions, the user can specify constraints on permissions, such as the number of times a permission can be used or the valid time per day during which a permission should be allowed. Regarding OCP detection, *Stowaway* [17] is solution for analyzing and detecting over-claimed permissions. *Stowaway* fuzes Android APIs directly, while *PScout* [4], another solution, fuzzes the applications which use the API. *PScout* finds the actual minimum set of required permissions according to application's API calls. *AdDroid, AdSplit and Leontiadis et al.'s work* [34, 39, 27] are dealing with advertising libraries, because they are themselves are vulnerable to OCP issues. *MockDroid* [6] and *AppFence* [25] restrict access to private data and reduce over-claimed permissions as well. *TISSA* [46], in addition, provide four options when dealing with private data – keep public, anonymize, provide fake result or provide empty result. *Flex-P* [30] allows not only to grant a subset of permissions at install-time, but also to change it at run-time.

The last category, Permission Escalation Attack, involves tracking and controlling the information flows through ICC between applications. Specifically, implementations use ICC chains to prevent PEA attacks like confused-deputy attacks or collusion attacks. All solutions address the confused deputy attack, but none of them fully addresses the collusion attack. However, XManDroid [9] and TaintDroid [12] partially address overt channel communication and XManDroid, in addition, partially address also covert channels communication in the collusion attack. The first representative, *TaintDroid* is dealing with PEA issue as well as OCP issue. It tracks information flows by labeling (tainting) data from privacy-sensitive sources. Afterwards, the user is alerted when any tainted data aim to leave the system at a taint sink. *XManDroid* (eXtended Monitoring on Android) tracks and analyzes the communication links among applications, and ensures that the application comply to a desired policy. Speciality of this implementation is making the policy decisions based on the content of intents. For example, application that is notified about incoming or outgoing calls and can record audio must not communicate to an application with network access. *QUIRE* [10] is lightweight provenance system primarily aimed to defend against the confused deputy attack. It also tracks and record the specific ICC call chain so that the recipient can observe the full call chain associated with a request. *QUIRE* lies on the middleware layer as well as on the kernel layer in order to be able to analyze remote procedure calls. The other solution, *IPC Inspection* [18] has been based on a similar principle – tracking information flows through ICC. The last compared representative in this category, *Sorbet* [19], allows developers to define policies to mitigate undesired information flows via tracking the permissions of all components on a call stack. Returning to figure 2.5, when the service S_A protected by permission P_1 is called by the S_3 which would be hypothetically called by S_1, Sorbet evaluates if every component on the call stack has the permission P_1 granted – so not only application 3 containing the service S_3, but also application 1 and their respective service S_1 is checked.

To supplement this enumeration and provide another categorization view, there exist

also Yuksel et al.'s comparison [45]. They classify all software-based solutions into four groups. These include operating system (OS) based, source code based and application or service based solutions and permission-based. The first category includes solutions which make changes on the operating system architecture. Studies under second category provide solutions by processing the byte-code of applications being based on analysis techniques or modification of byte-code and rebuilding the applications. The application or service-based examines the system as a background process or analysis is performed either manually or through the use of an application. The last group, permission-based solutions, aims to provide permission-based filtering and removes permissions that are considered to be harmful. Yuksel et al.'s team has performed assessment based on the following software-engineering criteria:

- **Overhead** – Does the solution cause overhead on the users' mobile devices?
- **Usability** – Is the proposed solution user-friendly?
- **Independency** – Is proposed solution device dependent?
- **Availability** – Is proposed solution available to end-users?

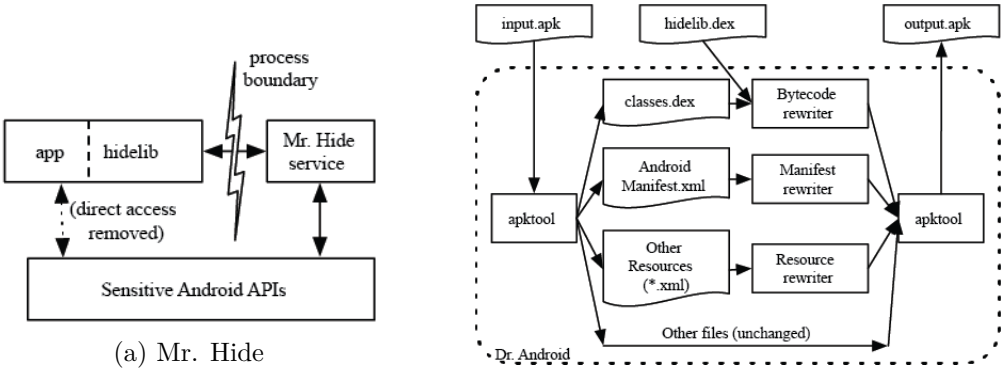
	Overhead	**Usable**	**Independent**	**Available**
Operating System Based	No	Yes	No	Yes
Permission Based	Yes	Yes	No	No
Source Code Based	Yes	No	No	No
Application Based	Yes	Yes	No	Yes
Service Based	No	Yes	Yes	Yes

Table 2.3: Findings of Yukse's et al. Assessment [45]

2.6 Jeon et al.'s Tools

A more detailed description of all compared frameworks is not possible because it is already beyond the scope of this work. However, it is beneficial to present detail of at least one solution in contrast to Aurasium, to obtain more critical view on the Aurasium framework itself. The chosen suitable framework is the bundle of Jeon et al.'s Tools. This is the most similar framework, which also does not require any modification to the Android platform. It includes *Dr. Android* and *Mr. Hide* tools by Jeon et al. In addition, *RefineDroid* supposed in this project supports these tools as analyzer tool. *Jeon et al.'s Tools* propose new permission model based on 7 selected most problematic coarse-grained permissions – `INTERNET`, `READ_PHONE_STATE`, `WAKE_LOCK`, `ACCESS_FINE_LOCATION`, `ACCESS_COARSE_LOCATION`, `WRITE_SETTINGS` and `READ_CONTACTS`. These permissions are replaced by the finer ones, which are however sufficient in most cases. For example, `READ_PHONE_STATE` is replaced by *Mr.Hide*'s permission UniqueID which returns randomly generated ID number (IMEI or IMSI) instead of whole state only [26]. Essential for this concept is, that new fine-grained permissions can be easily added over time. Tools are proposed to have different vocabulary of permissions for different applications and/or users, because everybody has different privacy requirements. In contrast to *Aurasium* project, *Jeon et al.'s Tools* are enforced in separate application process using the *Mr. Hide* application. Architecture of *Mr. Hide* is shown in figure 2.9a. Fundamentally, *Dr. Android* removes selected permissions and since

the Android's permission mechanism is robust enough to mediate also native code, it no longer has the access granted by the platform permission. Thus the application is rewritten to access these resources via *Mr. Hide* application which can perform mediation. Application is rewritten using the `apktool` similarly as in the *Aurasium*, however, *Dr. Android* also concatenates `hidelib.dex`, an adapter layer to connect to the *Mr. Hide* service. In *Dr. Android* must be rewritten also some other XML resources to support *Mr.Hide*'s `AdsPrivate` and `AdsGeo` permissions which replace coarse-grained `INTERNET` permission with permissions to display ads without sharing personal information. The process of repackaging is shown in figure 2.9b. Some shortcoming against *Aurasium* project is performance slowdown imposed by *Mr. Hide*, since the interprocess communication required by it is an expensive operation. The major drawback is, however, the availability of source code, since the only open-sourced tool is *Dr. Android* (`redexer`), and the required *Mr. Hide* part is used for developers' revenue. [26]



(a) Mr. Hide

(b) Dr. Android

Figure 2.9: Jeon et al.'s Tools Architecture [26]

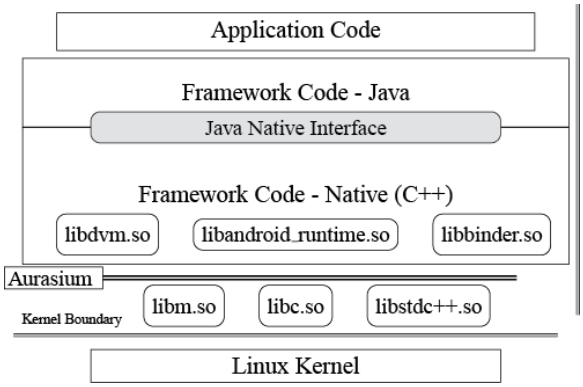
Chapter 3

Aurasium Framework

Aurasium project has been developed in 2012 as a project at the University of Cambridge, UK [44]. Aurasium uses repackaging mechanism, wrapping around the DVM under which the Android applications run, with monitoring code. It does not require rooted Android device. To attach sandboxing code, Aurarium exploits Android's unique application architecture of mixed Java and native code execution and introduces `libc` interposition code. Because of this, Aurasium is capable to mediate almost all types of interactions between the application and the Android OS.

This project consists of three parts – automated repackaging system written in Python programming language named `pyAPKRewriter`, monitoring code included in `ApkMonitor` application that intercepts an application's interactions with the system and `Aurasium's Security Manager` (ASM) application enabling central handling of policy decision of all repackaged application on the device [44].

Starting with sandboxing code, the top layer of the framework is written in Java. The aim is to create a well-documented easy-to-use abstraction layer upon cumbersome native layer of the framework. The upper layer creates interface for other possible programs and delegates all requests to the low-level part of the framework implemented in a native C++ code. This layer consists of few shared objects that do all the real work, such as communication with the Dalvik VM or establishing the mechanism for IPC communication. Figure 3.1a shows in detail the layers of the framework library in individual applications' *Logical Address Space* (LAS).



(a) Framework Structure

(b) Rapackaging System

Figure 3.1: Aurasium Architecture [44]

The second part of Aurasium, the repackaging Python script utilizes the previously mentioned sandboxing code and deploys it to Android APK installation package. APK file is similar to Java JAR archive and contains AndroidManifest file, application logic in the form of dex bytecode, compiled XML resources and native libraries. Each application package is also signed with authorship information. According to figure 3.1b, besides the sandboxing code, Aurasium has to include also several additional parts to APK in order to ensure the functionality.

The last part of the Aurasium is called *Aurasium Security Manager* (ASM). ASM handles the policy decisions centrally, so that all repackaged applications can be maintained at one place. Security policy is based on decision of application or user. Application decision works transparently without user interaction, while the user decision is consented by dialog window and can be remembered and used by default during next occurrence.

Since the project is introduced and publicly documented in [44] only superficially, the greater part of the information must be obtained by research or source code analysis.

3.1 Principle of Mediation

Aurasium mediation has been based on the interposition code of Android's standard C library called `libc` [44]. This library is called every time the upper layer framework library wants to interact with the OS itself. It is located directly upon the Linux kernel and initiates appropriate system calls into the kernel that completes the required operation.

The library is mapped in *Logical Address Space* of each process of every Android application using dynamic linking mechanism. Dynamic linking is maintained by C++ **ld.so** linker, which interconnects arranged compiled code of `libc` library with the framework libraries code in the LAS. In the Linux as well as Android OS, each compiled library located in the LAS, as well as a shared object file on the disk, is in the ELF format, so that the library could be shared and therefore mapped anywhere in the LAS, Position Independent Code (PIC) has been used. For this purpose, ELF object file dispose *Dynamic Symbol Table* (DST) in .dynsym section containing all of the file's imported and exported symbols used by linker to fill the pre-prepared read-write pages. Specifically, .got and .dynamic sections are used – .dynamic section is used for tagging the values during linking and the .god section contains Global Offset Table (GOT). GOT is used by stub functions in Procedure Linkage Table (PLT) to retrieve the real target address of the remote function. Since the Global Offset Table is located in the fixed distance from the text segment, instructions in the code can jump to correct GOT entry even if the library has been mapped in the arbitrary address. Firstly, the linker collects and maps all the libraries code and data into the LAS of process and after that, it fills the Global Offset Table with absolute addresses to ensure communication between modules and libraries. Overview of mapping into LAS is shown in figure 3.2. [35]

Therefore, Aurasium goes through every loaded ELF file and overwrites its GOT entries with pointers to its monitoring functions [44]. Functions themselves then mediate calls to actual library functions after they have completed monitoring, if necessary. Because there is no direct possibility to direct LAS modification using Java code, Aurasium implemented these interposition routines in C++, exactly as the `ld.so` loader is implemented. This is possible due to Android's *Native Development Kit* (NDK) and *Java Native Interface* (JNI) which ensures interaction between Java and native C++ code.

The mediation is used to dynamically monitor the application behaviour and enforce the finer-grained security policy. Aurasium introduces policies that protect the devices from

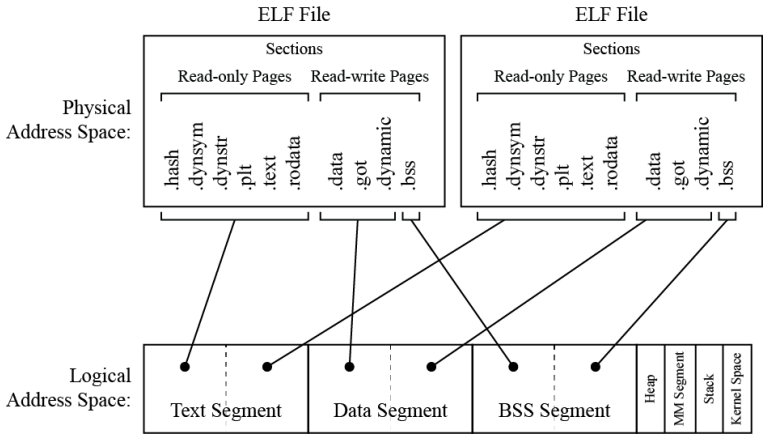


Figure 3.2: Mapping into Process's Address Space

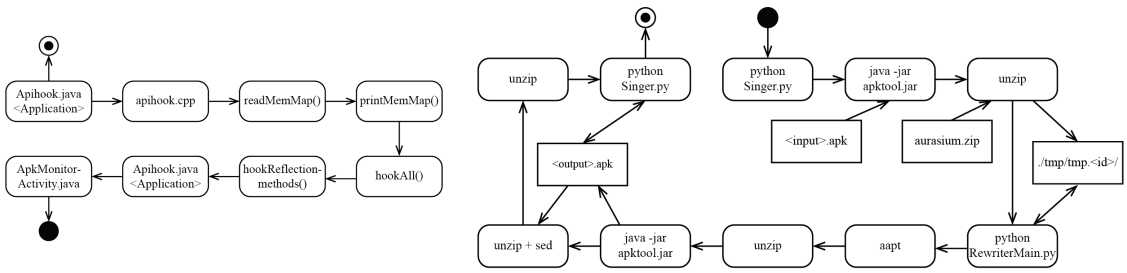
untrusted applications and their attempts to access sensitive information, leaking to the outside world or modifying it, to abuse SMS service or network connection as well as to escalate privilege and gain the root access [44]. All these refinements against standard built-in security policy in Android can be categorized into 3 categories – *Privacy Policy*, *Network Policy* and *Privilege Escalation Policy*. The purpose of the *Privacy Policy* is to enhance users' privacy. This is related to accessing the private data as the IMEI, IMSI, phone number, location, SMS messages, phone conversations or contact list [44]. The *Network Policy* enables finer-grained interaction with the network. For example, only particular web domains or set of IP addresses can be accessed. Furthemore, Aurasium proposes also IP blacklisting provided by the Bothunter [41] network monitoring tool to harvest information about malicious IP addresses. The last category, *Privilege Escalation Policy*, is used to secure the vulnerability introduced by Aurasium interposition.

3.2 Static Analysis

Aurasium's code is distributed under GNU General Public License and freely available on the GitHub server. Currently, analysis of the code is the only way to obtain more detailed information about this framework. The most important part of it is the native Android application called ApkMonitor. ApkMonitor contains all the sandboxing code, that is later attached to selected application which should be hardened.

The most profound part of ApkMonitor is written in C/C++ language and is called *Aurasium Native Library* (ANL). It contains two types of code – the code for preparing the interposition during startup and the code performing the mediation. The interposition code must be executed before any Android component is started so that the solution is robust enough. Android API prepares the `Application` component for this purpose, which is called before every component like Activity and can be used to include global initializations for an application. However, the most of the applications do not need to utilize this component, which is used in Aurasium. Aurasium must include this component (`ApiHook.java`) also in the `AndroidManifest` declarations. Since the implementation must rewrite the LAS of the application's process, it must be performed in the C/C++ language (`apihook.cpp`). The first operation is the analysis of LAS and reading of the memory sections (stored in `memmap` array). In the next step, each ELF file is accordingly mapped into `soinfo` structure, which

is used for patching and relocation of the addresses of `libc` functions to "hook" functions. A "hook" function (files starting with "`hook_`" prefix) represents the second type of native code – code performing the mediation. These function replaces the standard libc functions with mediation code and delegates further processing to lower layers in the end. This process is illustrated in figure 3.3a.



(a) Modified Implementation

(b) Rapackaging Script

Figure 3.3: Aurasium Static Analysis

Aurasium Native Library (ANL) must be supplemented by *Aurasium Java Code* (AJC) and new declarations in the AndroidManifest file. Aurasium tries to minimize the amount of native code, because it is generally difficult to write and test. For that reason, AJC contains all the policy logic and has been built upon many helper functions in the standard Android framework. However, including Java code into existing APK package is not trivial and requires some intermediate steps. In Android, all application Java code must be compiled to single file `classes.dex` which contains bytecode for DVM similarly to Java `Class` file, but contains all compiled Java files. Therefore, there is a need to disassembly the DEX file, insert Aurasium's sandboxing code and re-assembly it back to create the new `classes.dex` file. Fortunately, there exist open-source reverse-engineering tools to perform such tasks, which will be examined and overviewed in the following section. This tool is also used in Aurasium [11]. Regarding AndroidManifest.xml file, all components started in application must be declared in it and therefore, also modification of this file is part of repackaging the APK file. The principle of this approach is to attach Application class declaration in manifest file which will be instantiated by the runtime whenever the application is about to start. This enforces the GOT change (delegating the ANL) before any other parts of the original application run.

The second part of Aurasium, automated repackaging script, utilizes a combination of effective text processing of Python programming language and exploitation of Java and binary third-party console applications for Android development and hacking, which will be analyzed in next chapter in more details. All parts are interconnected by using Bash Unix shell interpreter as it is shown in figure 3.3b. In the first phase, the content of APK file is validated (using script `Singer.py`) and obtained using reverse-engineering tool `apktool.jar`. The next part is unzipping the ANL and launching the `RewriterMain.py` script, which injects the monitoring code. This essential script copies the ANL native code into `/jni` folder and adds new `<Application>` declaration in `AndroidManifest.xml`. In the next step, the APK file is packed again. Finally, the last part of the system is a Python script for the application singing since every application is required to have a valid signature. Signature in Android does not ensure the data integrity or confidentiality, but serves as proof of authorship. For example, user-defined permissions of signature protection

level type are granted automatically to the application packages with the same signature. Thus, Aurasium re-sign applications using a new self-signed certificate maintaining a one-to-one mapping between original certificates and equivalence classes of authorship among applications. Anyhow, this case is unique and Aurasium is usually applied to a standalone application where application updates and cooperation are not common.

Chapter 4

Analysis and Design

Apart from the theoretical outcomes, the aim of this thesis is to design the exemplary solution which would practically contribute to the security of Android platform. From the assignment, there are two main requirements – the first one defines the *way*, the second one defines the *goal*. Starting with the goal, the focus is aimed at the private user data and its restriction outside the device. The restriction should be performed without affecting the original behaviour of application, which implies dynamic policy enforcement and use of tainting mechanism, which will be later analysed in detail. Since this project is specific in its scope, hardening selected applications rather than whole system, restriction the data provision to other applications and system components become the most important. Due to the theoretical amount of work, the implementation is focused mainly on files and tracking its duplicates which could be also provided to sensitive Android components which are called before leaving the system. The data can be leaked through six types of media:

- **Data Cable** – Mobile device is connected and mounted to computer as USB device and data are leaked by copying.
- **Wi-fi** – Data are leaked through the socket interface using TCP/IP stack.
- **Bluetooth** – Data are shared using Bluetooth protocol similarily than Wifi.
- **Mobile Network** – Data are send via GSM *Teleservices* (standard call service), *Short Message Service* (SMS), or *Data Services* (GPRS, EDGE, UMTS).
- **Audiovisual Components** – Data are leaked through various output display and sound components of mobile devices and perceived through sensory organs or recorded using input devices (e.g. capturing the screen)
- **Other Techniques** – This includes stealing of physical device and its disassembling as well as other types of unusual acquisition.

The most risky media are the mobile and wireless networks, because they are the most probable way of leakage and they are also concern of this work.

The second part of requirements are the requirements which define the way which should be used in order to accomplish previously mentioned aims. The work should be built upon the Aurasium framework and accomplish the policy enforcement using the monitoring system calls. Android contains monitoring hooks for several system calls which are overviewed in the following table:

Network	IPC	System	File
connect()	ioctl()	dlopen() open()	fopen() read()
getaddrinfo()	close()	fork()	write()

Table 4.1: Intercepted System Calls in Aurasium

From this set, the system calls `ioctl()`, `open()`, `fopen()`, `read()` and `write()` were identified as suitable for this project. Starting with `read()` and `write()` functions, this functions are performing the reading and writing to file and their usage is straightforward. The functions are reading or writing from/to file identified with file descriptor (integer parameter) and using the memory identified by pointer (void * parameter). Functions use only limited memory defined by the third parameter. The system calls `open()` and `fopen()` are The next system call, `ioctl()`, performs a variety of control functions on STREAMS devices (defined by first parameter). Each device defines own set of supported commands, which are passed to this function as a second parameter. The following parameters are used without universal semantics and are also device-specific and contains additional data needed by certain device. Android uses this system call to implement its IPC, which is even more robust than standard Linux IPC and provides the communication between Android's components and is even used for various system events as touch control. Implementation is ensured by Android's Binder class and is analyzed in the next section in more details. In general, each remote call (between components) is mapped into corresponding call of ioctl() function. The data parameters uniquely define the destination class, called method and data and is provided in special format on the address specified by the third parameter, which will be also the subject of further examination. The searching of relevant classes and methods is objective of experiments with the original system.

In addition to this theoretical basis, which addresses the way and goal of project, there has to be performed also analysis of Android's and third-party console tools, which are essential part of the Aurasium project and must be considered as building blocks during the design of system.

Since a lot of information in this chapter is obtained during experiments, which are described as a part of development phase in *Software Development* chapter, research work in this chapter is interspersed in time with the practical work described in the next chapter.

4.1 Android Binder

In order to perform the required mediation, the part of Android middleware called the Binder needs to be rewritten. The Binder was originally developed under the name *OpenBinder* [2] by *Be Inc.* and later under *Palm Inc.* and provides high-level abstraction on the top of traditional modern operating system services including the facility to provide bindings to functions and data from one execution environment to another [37]. In Android, OpenBinder is customized to provide Inter-component Communication as has been described before. All interposition code needs to be placed in the suitable position in the original Binder implementation. Therefore, there is important to understand the concepts and to analyse the architecture of this part of system.

The communication between two processes is ensured using Binder Objects (BO), which are instances of classes that implement ioctl-based Binder interface. The most important operation which is declared in this interface is `transact(int code, Parcel data, Parcel reply, int flags)`. The corresponding callback method in the Binder object is called

`onTransact()`. The interface can be further extended by additional business operations using Android Interface Definition Language (AIDL) [20]. Each BO uses local and global identifier. The local ID is unique in the process and the global ID is created when the BO is passed to another process using Binder Driver (BD). The BD then works like network switch and persists the mapping from local ID to global ID in the table structure and translate it transparently, similarly than the mapping using ARP protocol. The Binder framework communication uses the client-server model. However, the process can implement the server, as well as the client, so the communication can be still bidirectional. The Binder Client (BC) invokes an operation on a remote Binder object called Binder Transaction (BT), which may involve sending or receiving data over the Binder Protocol. The communication is performed indirectly using Binder Driver. In the Android, the Binder Driver is exposed via `/dev/binder` file and simple API based on `open()`, `release()`, `poll()`, `mmap()`, `flush()` and `ioctl()` operations. Most communication happens via `ioctl(int fd, unsigned long request, ...` method. The first parameter is the file descriptor number which identifies currently open file and is used in `/proc/<pid>/fd/<fd>` file. The second parameter specifies the `ioctl()` command. The main commands are as follows [37]:

- `BINDER_WRITE_READ` – sends zero or more Binder operations, then blocks waiting to receive incoming operations and return with a result,
- `BINDER_SET_WAKEUP_TIME` – sets the time at which the next user-space event is scheduled to happen in the calling process,
- `BINDER_SET_IDLE_TIMEOUT` – sets the time threads will remain idle,
- `BINDER_SET_REPLY_TIMEOUT` – sets the time threads will block waiting for a reply until they time out, and
- `BINDER_SET_MAX_THREADS` – sets the maximum number of threads that the driver is allowed to create for that process's thread pool.

In fact, most communication happens via `ioctl(binderFD, BINDER_WRITE_READ, &bwd)` operation, where the binderFD is used to access the `/dev/binder` file and the `bwd` structure is defined as in figure A.1 and illustrated in figure 4.1.

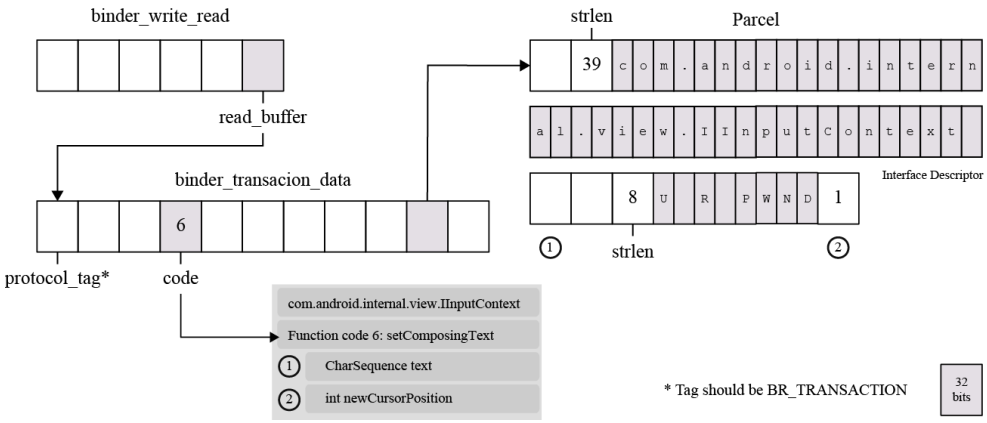


Figure 4.1: Semantics of `binder_write_read` Structure[1]

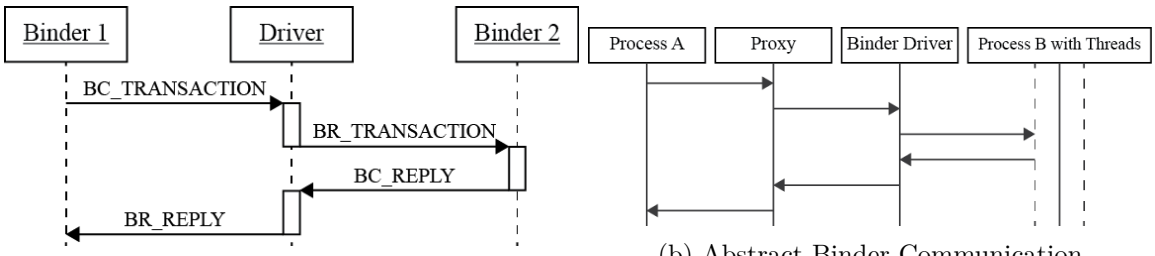
The `write_buffer` contains a series of commands for the driver to perform, while the `read_buffer` contains commands for the BO in user-space. The commands for driver are called Binder Call (BC) commands and the commands for the BO are called Binder Return (BR) commands. Each command is couple (operation code, data). These couples are stored in `binder_transaction_data` structure A.2. If the transaction is inline, the data is directly contained, otherwise, it contains a pointer to the data buffer.

The list of available commands, which can be stored in `write_buffer` respectively `read_buffer` is shown in following table 4.2:

write_buffer	read_buffer
BC_TRANSACTION, BC_REPLY, BC_ACQUIRE_RESULT, BC_FREE_BUFFER, BC_INCREFS, BC_ACQUIRE, BC_RELEASE, BC_DECREFS, BC_INCREFS_DONE, BC_ACQUIRE_DONE, BC_ATTEMPT_ACQUIRE, BC_REGISTER_LOOPER, BC_ENTER_LOOPER, BC_EXIT_LOOPER, BC_REQUEST_DEATH_NOTIFICATION, BC_CLEAR_DEATH_NOTIFICATION, BC_DEAD_BINDER_DONE	BR_NOOP, BR_TRANSACTION_COMPLETE, BR_INCREFS, BR_ACQUIRE, BR_RELEASE, BR_DECREFS, BR_TRANSACTION, BR_REPLY, BR_FAILED_REPLY, BR_DEAD_REPLY, BR_DEAD_BINDER, BR_ERROR, BR_OK, BR_ACQUIRE_RESULT, BR_FINISHED, BR_ATTEMPT_ACQUIRE, BR_SPAWN_LOOPER, BR_CLEAR_DEATH_NOTIFICATION_DONE,

Table 4.2: Binder Driver Commands

The Binder Transaction is a passing data from the client to the service, while the Binder Reply is a passing data from the service back to the client. This is shown in figure 4.2a. The whole Binder framework mechanism is transparent for the Android developer, since the Binder Transaction is performed as a local function call using so-called thread migration. This is ensured by the proxies and stubs, which are auto-generated helper classes from the AIDL files. The proxy, as it is illustrated in figure 4.2b, is the helper class which transforms Java code to low-level commands for the Binder Driver. The stub works in reverse to proxy and automatically parses and performs read commands on the service side. Since the Binder Driver is implemented on the low layer using C language, there has to be mechanism for encapsulation of high-level Java objects. This is ensured by `Parcel` container and corresponding `Parcelable` interface. A procedure for converting this higher-lever applications data structures into parcels is called marshalling. The marshalling, as well as unmarshalling, is also in the responsibility of the proxies and stubs.



(a) Binder Driver Interaction

(b) Abstract Binder Communication

Figure 4.2: Binder Architecture[2]

[1]Inspired by Artenstein et al.'s article [3]
[2]Inspired by Schreiber's article [37]

4.2 Command Line Tools

All Android projects and mainly this project, as well as original Aurasium framework itself, would not be possible without third-party tools. It includes developer tools provided as a part of the Android platform, but also the hooking, instrumentation and static analysis tools which are usually standalone open-source projects.

A detailed analysis of all existing tools is not possible, however, there has been identified and described some most important tools with regard to this project. *Android Debug Bridge*, a command line tool needed for development and especially for the communication with the physical or emulated device, is essential in this project, because the profound part of obtaining the information needed for design and implementation are experiments, which are performed due to this instrument. The second very important tool in this category is the *Device Monitor*, which enables tracking the file system and provides also graphical tools for debugging and analysis of Android applications. The part of Android SDK is also the third selected tool, *Android Asset Packaging Tool*, for managing the content of APK files. The next described tools are static analysis tools *Smali* and *Baksmali*, which are utilized in repackaging script for unpacking the Class files for injection. The last reverse-engineering tool, *Apktool*, is higher-lever utility which simplifies the use and wraps the other tools together. Information is based on Android Hacker's Handbook book [11] and on the manuals and documentations provided with the tools.

Android Debug Bridge

Android Debug Bridge (ADB) is universal tool for command line, which enables communication with emulator or the physical device with Android OS. This client-server application contains three parts:

- **A Client** – which runs on development machine and can be invoked from a shell by issuing an adb command.
- **A Server** – which runs as a background process also on the development machine and manages communication between the client and the adb daemon running on an emulator or device.
- **A Daemon** – which runs as a background process on each emulator or device instance [1].

When the ADB Client is started and server process is not already running, it starts the server process, which subsequently binds to local TCP port 5037 and listens for commands send from all adb clients. The actual communication between development machine and the physical or emulated device is performed between the ADB Server and Deamon using the odd-numbered ports in the range 5555 to 5585, the range used by emulators or devices. The even-numbered port is assigned to each device for console connections. ADB Server handles commands to all connected devices. The selection of some ADB commands is shown in table 4.3.

Device Monitor

Android Device Monitor (ADM) is a stand-alone tool that provides a graphical user interface for several Android application debugging and analysis tools. The Monitor tool does not

Category	Command	Description
General	devices	Prints a list of all attached instances.
	help	Prints a list of supported ADB commands.
Debug	logcat	Prints log data to the screen.
Data	install	Pushes an Android application to an instance.
	pull	Copies file from an instance to the computer.
	push	Copies file from the computer to an instance.
Networking	forward	Forwards socket connections from local port to a remote port on the instance
Server	start-server	Starts the ADB server.
	kill-server	Terminates the ADB server.
Shell	shell	Starts a remote shell in the target instance.

Table 4.3: Android Debug Bridge Commands [1]

require installation of an integrated development environment (IDE) [1]. ADM encapsulates the following tools:

- **Dalvik Debug Monitor Server (DDMS)** – debugging tool, which provides printscreen of device, informations about threads, state of RAM memory, incoming calls, SMS messages as well as file system.
- **Tracer for OpenGL ES** – tool for OpenGL Embedded Systems (ES) code analysis in Android application. This tool enables OpenGL ES commands and screen catching
- **Hierarchy Viewer** – enables debugging and optimization of GUI providing the visual representation of component hierarchy.
- **TracerView** – a graphical viewer for execution logs saved by your application, which helps to debug the application and profile its performance [1].

Smali and Baksmali

Smali (resp. *Baksmali*) is an assembler (resp. equivalent disassembler) for Dalvik Virtual Machine. DVM executes instructions represented in Dalvik bytecode and stored in Dalvik executable (DEX) format. This tool works similarly as the ordinary assembler for physical machine, but it converts files from Java assembler language called Smali to DEX format. Illustration of this principle is shown in figure 4.3. Smali syntax is based on *Jasmin* and *dedexer*. Jasmin is the standard assembly format for Java, *dedexer* is another DEX file disassembler like *Smali*, which is focuesd on Dalvik operation codes.

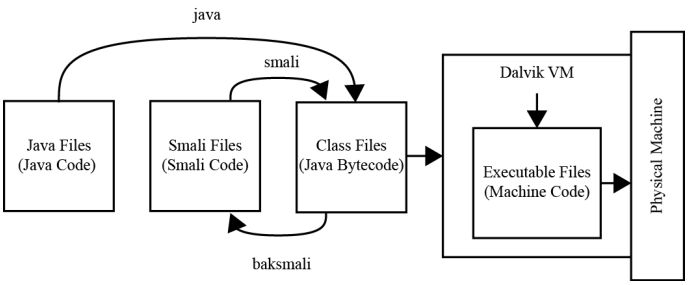


Figure 4.3: Smali Principle

Android Asset Packaging Tool

Android Asset Packaging Tool (AAPT) is a tool for managing the content of Android APK files. It can view, create, and update Zip-compatible archives (zip, jar, apk) and also compile resources into binary assets. This tool is the part of Android SDK used every time the developer wants to generate new APK file, since it is the base builder for Android aplications.

Apktool

Apktool is an universal Android reverse-engineering tool which also utilizes and wraps the *Smali* and *Baksmali* tool as well as AAPT. The tool can unpack the existing APK files into the original resources contained in them in human-readable XML form. It can also produce disassembly output of all classes and methods using Smali. Apktool creates project-like file structure and includes also embedded Smali debugger.

4.3 System Design

Design of the system is based on previous analysis and various experiments, which were focused on the Android system behaviour. Design consists of three parts – design of architecture and principle of application, design of data structures and design of configuration. Design of architecture can be further divided into two parts – design of tainting mechanism and design of restriction.

Starting with the overall architecture and tainting principle, the tainting is based on the principle used in TaintDroid architecture shown in next figure 4.4.

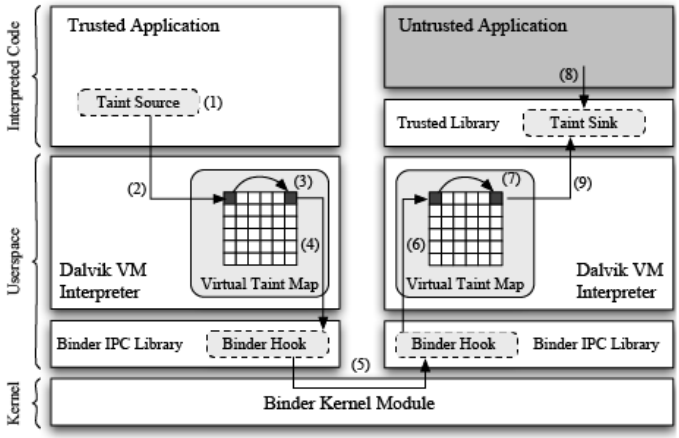


Figure 4.4: Tainting Principle [12]

In order to perform real memory-level tainting, there has to be tracked each atomic memory transfer, which from a programmer's point of view means that each assignment to a variable. This can be done only through monitoring of instructions on the level of a machine. In TaintDroid, there are monitored instructions on the level of virtual machines, because all possibly harmful applications are run under Dalvik virtual machine. TaintDroid uses *Virtual Taint Map* (VTM), which mirrors the address space, but does not contain the content of the memory. It represents the division of memory into a protected (black square

in figure 4.4) and a public (white square) part. Before tainting process, the tainted files are marked in VTM. Then, every copying of memory invokes copying of blocks in VTM. Since the applications, which run on separate DVM can also exchange data, TaintDroid introduces message-level tainting as well.

In this project, there have been designed and integrated two granularities of taint propagation – file-level tainting and data-level tainting. The message-level tainting (between components) principle from TaintDroid is used only for final policy enforcement (restriction), because Aurasium intercepts only single applications and does not have possibility to monitor the unhardened ones. File-level tainting and data-level tainting use the previously mentioned concept of VTM, but it is stored in higher-level abstract data structure and file instead of VTM.

File-level tainting between memory and the OS's file system can be performed in a full scope, because Aurasium can fully intercept this communication using system calls `fopen()`, `open()`, `write()` and `read()`. Function `fopen()` is used for obtaining the opening mode as is described in 5.2. This is used for designed *Tainting Customization* (TC). If the untainted memory is written to tainted file in append mode, the files remain tainted, but if it is written in read mode, the file becomes untainted. The open() and read() calls are used for tainting the memory blocks as well as new files. The data in memory read from tainted file are marked similarly and the files, which are read from tainted memory blocks become tainted too. However, data in memory are also directly propagated.

Since the Aurasium can intercept only specific places (system calls) and not instruction itself, it is impossible to implement full-scope memory-level tainting as is introduced by TaintDroid. This is replaced by the newly designed data-level tainting concept. This concept together with the file-level tainting is depicted in figure 4.5. When the data are read from the file, the content of data is read and tagged using a hash function which assigns a unique number. This tag, together with the size of block is used during the writing unknown memory block into the file. Each unknown memory block is tested with respect to any existing hash and marked as tainted if the hash matches. Subsequently, the file is marked as tainted as well.

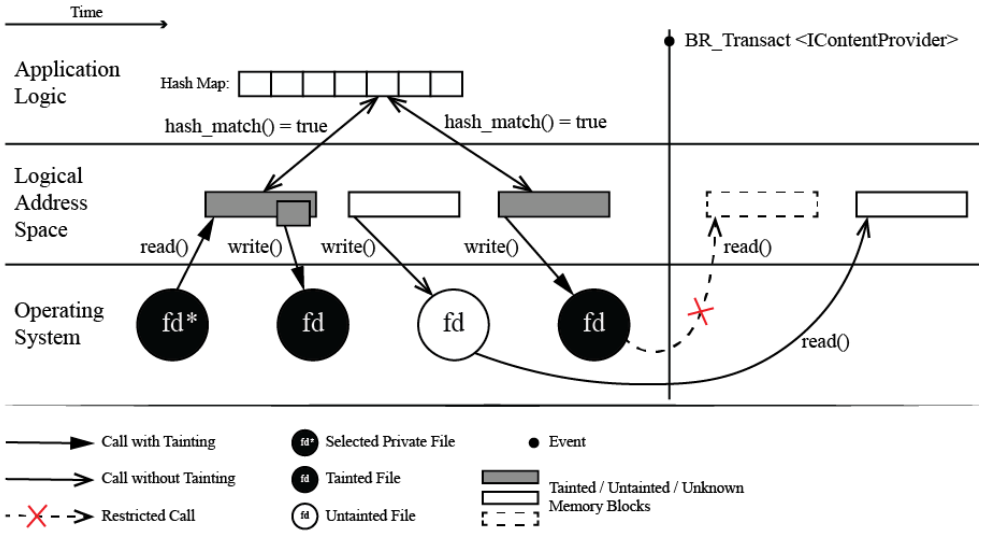


Figure 4.5: Design of Architecture

The final policy enforcement is performed using the interception of `ioctl()` call. Specif-

ically, when the `BR_TRANSACTION` command which contains destination component ContentProvider is read, all the read() calls for the tainted files are in the mode of restriction. The project is proposed to secure the user-selected files or folders as an entity, which are intended to be invariable like images, pictures or videos. Documents that are often changed can be restricted for opening, or there can be assigned unique rights for opening to hardened application and the files are encrypted for other applications. In this Inverse Mode (IM), data is protected with unhardened applications and uncovered and possibly exploited by the hardened application. The inverse mode is designed as optional and may not be implemented. A further extension is finer granularity of data-tainting. Unknown memory block which is being written to file is compared against the tainted memory blocks which are smaller than the unknown memory block, and the memory block which is being read from file has been divided into smaller units with separate hash.

Data structure which works as TaintDroid's VTM is designed as a simple array of memory blocks, which is the interconnection part between the file system level and application logic performing described data-tainting. From designed perspective, each memory block is considered as tainted or untainted. Application can store only tainted data and other will be implicit. Initially, the user-selected files are marked as tainted and during the tainting process, the other files and new memory blocks are added. Each memory block has assigned only one file which is the source of its data, one counted hash tag, but a lot of destination files to which this data is written. Due to TC, also the file modes need to be stored, because read() and write() functions do not dispose with this information in the passed arguments. The figure A.3 shows the pseudocode of the designed data structures for this tainting process.

Regarding the configuration possibilities, the main purpose is to allow the selection of private files and folders. Since the two previously-mentioned tainting mechanisms are preferred in different situations, the setting of selected mechanism is also appropriate. Hash-only tainting can be used in situations where the low memory consumption is important, while the content-based scanning in the case of files which need better protection. In some cases, there is also a need to disable the tainting completely due to performance slowdown. The restriction can be performed explicitly as well as dynamically. Explicit (static) restriction can be used in situations which require protection from theft or unauthorized users. One exemplary utilization includes parental control. Dynamic enforcement can be realized using confirmation screen and is useful for its flexibility. It is also appropriate to consider the configuration of permanent restriction where the selected protected files can not be even read by hardened application. The configuration settings can be assigned to each application separately or centrally. GUI can be also resolved as a single central application which communicates which hardened applications or as injected screen in each application. For the purposes of this project, the first variant is chosen. The advantage is an easier configuration for multiple applications at once, usability and better overview of the whole configuration in one place. However, there is security issue related to the communication between configuration application and the hardened ones. The communication is designed via configuration file for its simplicity. This file should be secured to ensure mainly the data integrity. Returning back, the usage with the permanent restriction, especially as a parental control, requires the password protection of configuration application itself.

CHAPTER 5 IS NOT PUBLISHED

Chapter 6

Verification

The testing of application is distributed across the various phases of the software development. There are used different approaches on the different testing hierarchy levels based on these phases. Since this thesis is not interested in a creating of a commercial product, there are not performed acceptance tests with the third-party persons. On the other hand, there are conducted mainly the developer-side tests – unit tests and assembly tests, as well as the black-box tests – function tests and system tests as is shown in figure 6.1.

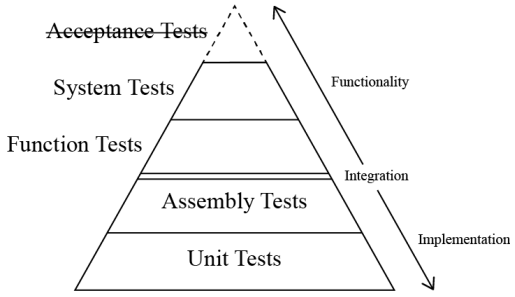


Figure 6.1: Peformed Tests According to Hierarchy Level

Since the application implementation is rooted in a deep theoretical knowledge of Aurasium framework as well as the principles of Android platform on the lowest level, the essential fundament of verification is theoretical assessment and evaluation of the testing outputs and the overall project outcomes.

6.1 Testing

In the integration phase, there is tested standalone Aurasium framework functionality against the several Android versions before and after debugging and porting to newer Android version. The chosen methodology is developing the simple testing application which reads and writes to file. Then, there are evaluated the results from intercepted system calls with the obtained theoretical knowledge. The next phase, experiments, is concerned with the assembly testing with the real selected applications – *IO File Manager* and the *Ted Text Editor*. There is tested system calls monitoring of Aurasium Framework in a real environment. However, the main purpose is to pre-prepare and verify the testing environment itself for the subsequent testing of implemented modifications. In the further phase – implementation – there are performed assembly tests with the developed mocking

environment. The mocking environment is implemented as the simple standalone C++ application with the several predefined test scenarios. Test scenarios are sequences of system calls inserted to the testing environment using `#include` directive and designed according to performed experiments which are intended to test a specific part of the tainting mechanism. The system output, as is outlined in a previous figure 5.2, can be performed also in this environment.

The implemented hook functions have been integrated into the real environment – firstly into the simple testing application which is only capable of reading and writing to file, then in a pre-prepared mocking environment with the two chosen publicly-available open-source applications – *IO File Manager* and *Ted (Text Editor)*. The testing has been carried out by designed logging to files. Especially, the content of `pe_log_taint_map.txt` has been important, because there is tracked the content of all tainting structures in time, which is the only output of tainting process. The final restriction is tested performing the black-box function test with the manually repackaged application *IO File Manager* and *Ted Text Editor*. Manual repackaging of applications is important during the testing phase, because the automated repackaging script is working on the level of smali assembler and creates APK installation files instead of application source code which is compilable and testable using e.g. debugging techniques. In *OI File Manager* has been tested tainting mechanism, final restriction using two different methods and communication with the configuration activity. Regarding tainting mechanism, there has been performed actions "move", "copy", "rename", "open" as well as "send" operation in order to test the proposed test scenarios in real hardened application. Action "delete" is not considered in the system, but is also taken into account for future improvements – e.g. disabling its operation on tainted files. The second application, *Ted (Text Editor)*, is used for testing of explicit restriction. There are conducted three scenarios – opening the unprotected file, opening the protected file in empty data falsifying mode and opening the protected file in fake data falsifying type.

The configuration application, as well as overall functionality of the system, has been consequently tested utilizing the manual system testing. There has been proposed several test cases and their expected outcomes and tested accordingly. Especially, the application has been tested with regard to sharing data using Bluetooth, Wi-fi, SMS, Email and other channels [B.5]. The outcomes of this testing in OI File Manager Application is illustrated in table 6.1.

All sharing methods have been tested on various types of media files – text files, images, animated GIFs and audio files. As regards the sharing via email clients, the application has been tested on two different clients, each of them behave correspondingly. The standard Email application from Google Inc. is not treated for the interception and is not started when the restriction is considered, whilst the Email application from LG company handles the unexpected results from `query()` function and is started with the empty attachment [B.7]. However, in both cases, the private data is protected. A special case is the MLO Task Management Tool application, which does not share the data immediately, but stores the attachment in a local database. However, since the application added the synchronization between multiple devices, the data are also sent via network later. However, the application is restricted at the start, so the restriction is also effective. All the tests have been carried out on the real Android device running the hardened applications compiled with Android 4.3.1 build target. The testing device runs on Android version 4.1.2. The testing using Android Virtual Device (AVD) or utilizing the Genymotion® [21] Android Emulator is limited, because the sharing possibilities are lacking standard applications as well as the resources – mobile network connection or Bluetooth adapter. The tests have also been performed on

Sharing Method	Application	Outcome without Restriction	Outcome after Restriction
Email	Gmail	Email is sent	Application is not started
	Email by LG	Email is sent	Application is started with empty attachment
Bluetooth	Bluetooth	File is sent	Sending failed on sending device
SMS	Messaging by LG	SMS is sent as MMS	Application is not started with image attachment. Runs with text attachment, because it cannot handle it even without restriction
IM	Google+	Message is sent	Application is not started
	Hangouts	Message is sent	Application is not started
Other	MLO 2	Files is added to list	Application is started with empty attachment
	FX File Manager	File is saved in another location	Application is not started

Table 6.1: Testing of Final Restriction in OI File Manager Application

the OI File Manager application repackaged using automatic Aurasium script resulting with the expected isomorphic behaviour. The application GUI responds as expected and the private data are protected in all the cases.

Since the repackaging script is a part of Aurasium project itself, the testing of repackaging mechanism gives the isomorphic results as expected. The following table 6.2 adapted from Xu et al.'s article shows the overall evaluation.

Type of Application	Number of Applications	Repackaging Success Rate
App store corpus	3491	99.6% (3476)
Malware corpus	1260	99.8% (1258)

Table 6.2: Repackaging Evaluation Results [44]

Repackaging introduces only negligible part of the code in Java and the resulting added size of the introduced C code – 46.4 KB – is not relevant with regard to the large libraries and another code already included in original Aurasium implementation, the size increase after the repackaging corresponds to the original Aurasium implementation as is shown in figure 6.2.

With regard to the performance evaluation, tests have been conducted on real Android device *LG L9 (P760)* with 1 GHz processor, 1 GB of system memory and 4 GB internal storage. Startup time of hardened applications is considerably changed. Aurasium's overwriting of Global Offset Table entries is consuming operation and lasts approximately 10 s as is shown in following table 6.3.

Application	Original (ms)			Repackaged (ms)			Time	Slowdown (ms)
Ted (Text Editor)	1016	1056	1184	11187	11040	11236	10x	10069
OI File Manager	729	830	645	11132	10238	10694	15x	9953

Table 6.3: Startup Time Slowdown on Repackaged Applications

The performance of the selected actions has been tested on the original application before repackaging, on the repackaged application with the inactive tainting and restric-

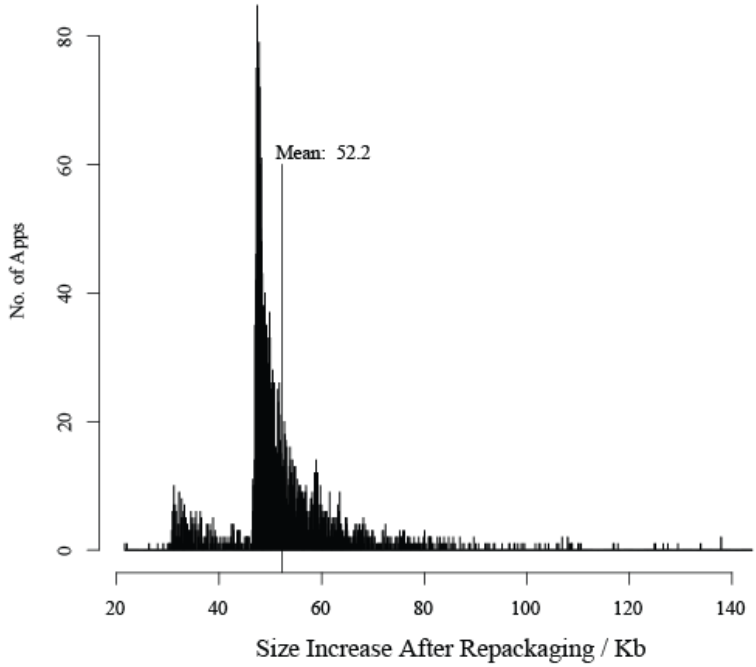


Figure 6.2: Size Increase After Repackaging in Original Aurasium [44]

tion and on the repackaged application with the active protection. The aim of this tests is to identify application performance from the user perspective and detect unexpected performance behaviour which could also lead to the discovery of hidden error and covert vulnerabilities. In order to determine real performance, all logging outputs are disabled. Since the time has been measured using logging outputs as well as by manual measurement, it is measured repeatedly in order to estimate deviations and accuracy. There is also calculated overhead of the repackaged application with active functional protection. The performance of restricting has been tested on two actions – sending via email and sending via Bluetooth, as is shown in table 6.4.

Action	**Method**	**Original (ms)**			**Inactive (ms)**			**Active (ms)**			**Gain**
Send via Email	Mediation	1079	1047	922	1469	1258	1515	1320	1368	1336	32%
	File Protection							1226	1297	1351	27%
Send via Bluetooth	Mediation	445	382	508	992	953	914	985	961	1000	121%
	File Protection							1144	1047	1063	144%

Table 6.4: Performance of Sharing Actions with Restriction

The times are subtracted from standard Android log messages. The overhead in Bluetooth application has been significantly bigger than Email application, which has been caused by different implementation and reaction on unexpected condition caused by introduced restriction. As is shown in the previous table, the time of processing is more dependent on the application which handles the action, than the method of restriction.

The tainting performance has been tested for combinations of scanning type and protection of copied file regarding configuration. Specifically, there is measured a duration of paste action between the start and end of the copy process. The results in table 6.5 show limitations of this test, because slowdown is too small to be obtained by this method.

However, since the test is aimed at user experience, the result is satisfactory, because the performance of the hardened application remains almost the same even during the active protection.

Action	Scanning	Original (ms)			Inactive (ms)			Active (ms)			Gain
Untainted Copying	File-based	2394	2479	2421	2675	2643	2647	2787	2608	2778	12%
	Content-based							2665	2784	2586	10%
Tainted Copying	File-based							2707	2626	2596	9%
	Content-based							2792	2735	2663	12%

Table 6.5: Testing of Tainting Performance

The last test is focused on the performance of file opening and reading during increased threat level mode. The worst results are obtained during the enabled protection with fake data falsifying, because the data must be overwritten in system memory [tab. 6.6]. Empty data falsifying protection is faster compared to the previous method, but still represents about 20% overhead against the unhardened application.

Action	Data Type	Original (ms)			Inactive (ms)			Active (ms)			Gain
File Opening	Empty Data	736	732	704	1881	1425	1357	948	859	816	21%
	Fake Data							1901	1438	1513	23%

Table 6.6: Time of File Opening During Increased Threat Level

To summarize the performance test results, the startup time overhead of repackaged applications is the biggest performance drawback. Otherwise, there can be seen some slowdown, but it is nearly transparent to the user and does not represent an obstacle to the use. The average slowdown is, in most cases, similar to Aurasium's proposed intercepted actions in [44].

6.2 Evaluation

The overall system can automatically repackage the selected applications and then dynamically enforce the security policy for sharing selected files. Nevertheless, the unrelated operation and functionality of the hardened application remains unchanged and can be utilized as before. The application offers simple GUI for selection of private files and configuration of desired behaviour or stopping the security scanning completely. The main benefit is the security refinement. Aurasium mediation is reliable in the most cases due to sophisticated monitoring of system calls on the layer lower, than application layer used by most programmers.

In order to achieve these results, there has been researched the principles of operating systems, primarily Linux OS, the principles of Android OS from the security perspective and examined and compared several frameworks with the similar functionality than the analyzed the Aurasium framework. After that, the Aurasium framework has been critically studied in detail and analyzed its code due to insufficient documentation. The Aurasium has been documented and tested its functionality on the older Android API platform. Subsequently, it has been mapped to newer Android version and investigated automatic repackaging script in Bash and Python language. After that, there has been proposed the first application design and chosen the suitable publicly-available application. Then, the applications have been manually repackaged and the experiments upon them have

been performed. Subsequently, the proof of concept has been implemented performing the restriction mechanism. After that, the overall solution has been designed. The next step was developing the mocking environment and implementation and testing the tainting mechanism. This has been followed by integration in a real environment – the simple testing application, and the real open-source applications. Consequently, the configuration activity has been developed and tested. Lastly, the overall system has been tested.

The hardest part of this process is the understanding of the Aurasium design without documentation as well as the collecting and associating the information from a number of sources addressing various disciplines as operating systems, Android platform, compilers and dynamic linking, various existing software (Aurasium and other frameworks) and tools (console tools, projects utilized in solution itself). A necessary prerequisite is also the understanding and analysis of code written as in native-based C/C++ language as in Java bytecode and scripting languages Bash and Python. The biggest challenge in the project is also the finding the design of resulting solution as well as the finding the path in order to realize this design using limited resources – the lack of documentation, the lack of theoretical background, limited building blocks, limited time and other shortcomings like bugs in Aurasium or shortcomings in development environment (inability to debug native code in repackaged applications).

The main benefit is also in the theoretical outcomes and the design of solution which exceeds the actual implementations in the scope and a number of proposed extensions. The personal benefit of this work is in better orientation and understanding of code programmed by another developer. Aurasium framework, as well as other frameworks and tools, provides interesting solutions to the general problems and design patterns which can be reused in other projects. The benefit is also greater ability to interconnect and reuse these ready-made solutions. Another theoretical outcome is understanding of all-pervasive communication using Android binder in Android OS as well as the other principles of Android OS from the lower perspective and operating systems in general. From the programmers perspective, biggest asset is the understanding Java JNI principles and possibilities and ability to develop a native-based application in Android platform. The more general benefit is also the ability to independently define, solve and assess the problems. There has also been designed a number of extensions for further development in chapter 4.

Chapter 7

Conclusion

The aim of this work was to provide the system for securing the user-selected private data of chosen applications with the sandboxing mechanism. In the first phase, interprocess communication and existing frameworks, which are capable of intercepting communication between the application and the operating system on the level of system calls, are explored. Subsequently, the possibilities and the code of the one of the compared frameworks – Aurasium framework – are investigated. From this analysis, the framework has been modified to be able to build and proper run on the required Android version, and there were conducted experiments in order to propose and realize the tainting mechanism as well as the security policy enforcement.

The system performs the file-based tainting of selected files using hash calculation and content-based tainting using division of file content into "small blocks". The restriction is implemented intercepting the communication with an external application using `ContentProvider` interface and via falsifying the file content. There is also developed restriction based on falsifying the whole files on the file system. Application introduces communication with the configuration application utilizing the configuration file, which is similar than low-level Android binder communication. Configuration options include the type of scanning, type of restriction, method of intercepting or static restriction of file opening.

The system has been developed and tested in three different types of environment – in mocking C++ environment, in selected hardened test applications and in Aurasium repackaging system. The C++ environment is prepared on Linux machine with several test scenarios, the hardened applications are mostly tested on the real Android device using the Android IDE and the Aurasium repackaging system is tested by replacing the monitoring code in the original *ApkMonitor* package.

This work investigates the security problems in Android and significantly contributes to understanding its security architecture. There are aggregated principles from different disciplines and designed multiple figures to facilitate explanation. The work has also been awarded at Excel@FIT 2016 [8] student conference for the independent scientific approach and the elaboration quality. It is also chosen between 5 best scientific contributions and is about to be printed in SERSC journals [38].

The next steps are the code optimization, design, and implementation of the secure communication between configuration activity and the hardened applications, addressing and handling the file permissions or the implementation of several proposed extensions. The next work can be also aimed at developing the automated configuration and risk analysis, unique handling the files according to the type of media (image, video, text files), expanding

the configuration options, focusing on the possibilities of the Linux core or further testing, experimenting and research.

Bibliography

[1] Android: Android Developers. [online], 2016, [cit. 2016-01-26].
URL http://developer.android.com/

[2] Angry Red Planet: OpenBinder. [online], 2016, [cit. 2016-01-21].
URL http://www.angryredplanet.com/~hackbod/openbinder/docs/html/

[3] ARTENSTEIN, N.; REVIVO, I.: Man in the Binder: He Who Controls IPC, Controls the Droid. In *Europe BlackHat Conf*, 2014.

[4] AU, K. W. Y.; ZHOU, Y. F.; HUANG, Z.; et al.: Pscout: analyzing the android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security*, ACM, 2012, pp. 217–228.

[5] BANURI, H.; ALAM, M.; KHAN, S.; et al.: An Android runtime security policy enforcement framework. *Personal and Ubiquitous Computing*, volume 16, no. 6, 2012: pp. 631–641.

[6] BERESFORD, A. R.; RICE, A.; SKEHIN, N.; et al.: MockDroid: trading privacy for application functionality on smartphones. In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*, ACM, 2011, pp. 49–54.

[7] BISHOP, M.: *Introduction to computer security*. Addison-Wesley Boston, MA, 2005.

[8] Brno University of Technology: Excel@FIT 2016. [online], 2016, [cit. 2016-05-15].
URL http://excel.fit.vutbr.cz/

[9] BUGIEL, S.; DAVI, L.; DMITRIENKO, A.; et al.: Xmandroid: A new android evolution to mitigate privilege escalation attacks. Technical report, Technische Universität Darmstadt, 04 2011.

[10] DIETZ, M.; SHEKHAR, S.; PISETSKY, Y.; et al.: QUIRE: Lightweight Provenance for Smart Phone Operating Systems. In *USENIX Security Symposium*, volume 31, 2011.

[11] DRAKE, J. J.; LANIER, Z.; MULLINER, C.; et al.: *Android Hacker's Handbook*. John Wiley & Sons Inc, reprint edition, 2014.

[12] ENCK, W.; GILBERT, P.; HAN, S.; et al.: TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, volume 32, no. 2, 2014: p. 5.

[13] ENCK, W.; ONGTANG, M.; MCDANIEL, P.: Mitigating Android software misuse before it happens. In *Proceedings of the 15th ACM conference on Computer and communications security*, Citeseer, 2008.

[14] ENCK, W.; ONGTANG, M.; MCDANIEL, P.: On lightweight mobile phone application certification. In *Proceedings of the 16th ACM conference on Computer and communications security*, ACM, 2009, pp. 235–245.

[15] ENCK, W.; ONGTANG, M.; MCDANIEL, P.: Understanding Android Security. *IEEE Security & Privacy*, volume 7, no. 1, 2009: pp. 50–57.

[16] FANG, Z.; HAN, W.; LI, Y.: Permission based Android security: Issues and countermeasures. *Computers & Security*, volume 43, 2014: pp. 205–218.

[17] FELT, A. P.; CHIN, E.; HANNA, S.; et al.: Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*, ACM, 2011, pp. 627–638.

[18] FELT, A. P.; WANG, H. J.; MOSHCHUK, A.; et al.: Permission Re-Delegation: Attacks and Defenses. In *USENIX Security Symposium*, volume 30, 2011.

[19] FRAGKAKI, E.; BAUER, L.; JIA, L.; et al.: Modeling and enhancing android's permission system. In *Computer Security–ESORICS 2012*, Springer, 2012, pp. 1–18.

[20] GARGENTA, A.: Deep dive into Android IPC/Binder framework. In *AnDevCon: The Android Developer Conference*, 2012.

[21] Genymotion: Genymotion – Fast And Easy Android Emulator. [online], 2016, [cit. 2016-01-26].
URL https://www.genymotion.com/

[22] Google Inc.: Chromium OS - The Chromium Projects. [online], 2016, [cit. 2016-01-26].
URL https://www.chromium.org/chromium-os

[23] Google Inc.: Google Play. [online], 2016, [cit. 2016-01-26].
URL https://play.google.com/store

[24] HANÁČEK, P.; STAUDEK, J.: *Bezpečnost informačních systému*. Úřad pro státní informační systému, 2000.

[25] HORNYACK, P.; HAN, S.; JUNG, J.; et al.: These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In *Proceedings of the 18th ACM conference on Computer and communications security*, ACM, 2011, pp. 639–652.

[26] JEON, J.; MICINSKI, K. K.; VAUGHAN, J. A.; et al.: Dr. Android and Mr. Hide: Fine-grained security policies on unmodified Android. Technical report, University of Maryland, 2011.

[27] LEONTIADIS, I.; EFSTRATIOU, C.; PICONE, M.; et al.: Don't kill my ads!: balancing privacy in an ad-supported mobile application market. In *Proceedings of the Twelfth Workshop on Mobile Computing Systems & Applications*, ACM, 2012, p. 2.

[28] Linux Foundation: Tizen. [online], 2016, [cit. 2016-01-26].
URL https://www.tizen.org/

[29] Mozilla Foundation: Firefox OS. [online], 2016, [cit. 2016-01-26].
URL https://www.mozilla.org/en-US/firefox/os/

[30] MUELLER, K.; BUTLER, K.: Flex-P: flexible Android permissions. In *IEEE Symposium on Security and Privacy, Poster Session*, 2011.

[31] NAUMAN, M.; KHAN, S.; ZHANG, X.: Apex: extending android permission model and enforcement with user-defined runtime constraints. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, ACM, 2010, pp. 328–332.

[32] ONGTANG, M.; MCLAUGHLIN, S.; ENCK, W.; et al.: Semantically rich application-centric security in Android. *Security and Communication Networks*, volume 5, no. 6, 2012: pp. 658–673.

[33] Open Handset Alliance: Android Overview. [online], 2016, [cit. 2016-01-26].
URL http://www.openhandsetalliance.com/android_overview.html

[34] PEARCE, P.; FELT, A. P.; NUNEZ, G.; et al.: Addroid: Privilege separation for applications and advertisers in android. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, ACM, 2012, pp. 71–72.

[35] PRESSER, L.; WHITE, J. R.: Linkers and loaders. *ACM Computing Surveys (CSUR)*, volume 4, no. 3, 1972: pp. 149–167.

[36] ROSENBLATT, J.: Google's Android Generates 31 Billion Dollars Revenue, Oracle Says. [online], 2016, [cit. 2016-01-21].
URL http://www.bloomberg.com/news/articles/2016-01-21/google-s-android-generates-31-billion-revenue-oracle-says-ijor8hvt

[37] SCHREIBER, T.: *Android binder*. Master's thesis, Ruhr-Universität Bochum, 2011.

[38] SERSC: Journals. [online], 2016, [cit. 2016-05-20].
URL http://www.sersc.org/journals/

[39] SHEKHAR, S.; DIETZ, M.; WALLACH, D. S.: *Separating Smartphone advertising from applications*. Ph.D. thesis, RICE UNIVERSITY, 2012.

[40] SILBERSCHATZ, A.; GALVIN, P. B.; GAGNE, G.; et al.: *Operating system concepts*, volume 4. Addison-Wesley Reading, 9th edition, 1998.

[41] SRI International: BotHunter Internet Distribution Page. [online], 2016, [cit. 2016-04-26].
URL http://www.bothunter.net/

[42] TANENBAUM, A. S.; BOS, H.: *Modern operating systems*. Prentice Hall Press, 2014.

[43] VANČO, M.: *Presentation Manager for Android*. Master's thesis, Brno University of Technology, 2012.

[44] XU, R.; SAÏDI, H.; ANDERSON, R.: Aurasium: Practical Policy Enforcement for Android Applications. In *USENIX Security Symposium*, 2012, pp. 539–552.

[45] YUKSEL, A. S.; ZAIM, A. H.; AYDIN, M. A.: A Comprehensive Analysis of Android Security and Proposed Solutions. *International Journal of Computer Network and Information Security (IJCNIS)*, volume 6, no. 12, 2014: p. 9.

[46] ZHOU, Y.; ZHANG, X.; JIANG, X.; et al.: Taming information-stealing smartphone applications (on android). In *Trust and Trustworthy Computing*, Springer, 2011, pp. 93–107.

Appendices

List of Appendices

A Pseudocodes **61**

B Application Screenshots **63**

C DVD Content **65**

Appendix A

Pseudocodes

```
1  struct binder_write_read {
2    signed long write_size; /* Bytes to Write */
3    signed long write_consumed; /* Bytes Consumed by Driver */
4    unsigned long write_buffer;
5    signed long read_size; /* Bytes to Read */
6    signed long read_consumed; /* Bytes Consumed by Driver */
7    unsigned long read_buffer;
8  };
9
```

Figure A.1: Pseudocode of `binder_write_read` Structure

```
1  struct binder_transaction_data {
2    union {
3      __u32 handle; /* Target Descriptor of Command Transaction */
4      binder_uintptr_t ptr; /* Target Descriptor of Return Transaction */
5    } target;
6    binder_uintptr_t cookie; /* Target Object Cookie */
7    __u32 code; /* Transaction Command */
8
9    /* General Information about the Transaction. */
10   __u32 flags;
11   pid_t sender_pid;
12   uid_t sender_euid;
13   binder_size_t data_size; /* Number of Bytes of Data */
14   binder_size_t offsets_size; /* Number of Bytes of Offsets */
15
16   /* Transaction Data */
17   union {
18     struct {
19       binder_uintptr_t buffer; /* Transaction Data */
20       binder_uintptr_t offsets; /* Offsets to flat_binder_object Structs */
21     } ptr;
22     __u8 buf[8];
23   } data;
24 }
```

Figure A.2: Pseudocode of `binder_transaction_data` Structure

```
class SmallBlock {
  unsigned long start;
  char block[SMALL_BLOCK_SIZE];
};

class MemBlock {
    unsigned long start; /* Start Address */
    int size; /* Size of Memory Block */
    int fdSrc; /* Source File Descriptor */
    TaintedFile fpSrc; /* Full Path of Source File */
    unsigned char hash[32]; /* Counted Hash */
};

std::vector<MemBlock *> taintMap; /* List of Tainted Memory Blocks */

std::vector<SmallBlock *> smallBlocks; /* Blocks of Stored Information */


```

Figure A.3: Pseudocode of Data Structures for Tainting

Appendix B

Application Screenshots

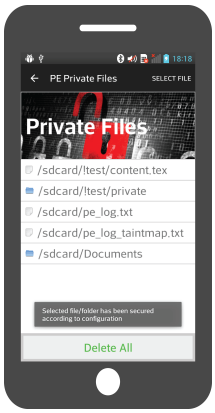


Figure B.1: Main Screen of Configuration Activity

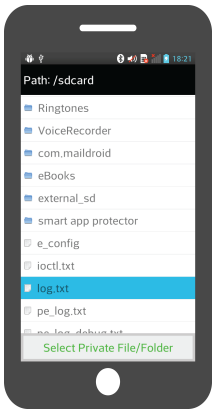


Figure B.2: Selection of Private Files

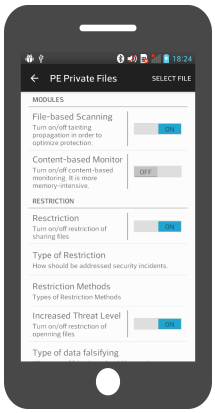


Figure B.3: Settings in Configuration Activity

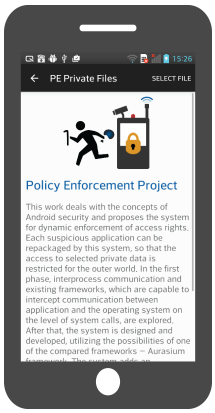


Figure B.4: Information Screen in Configuration Activity

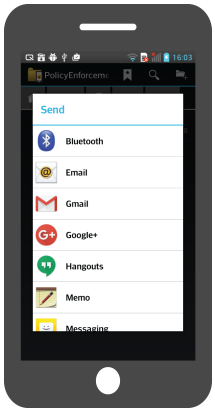


Figure B.5: Sample of Sharing Possibilities

Figure B.6: Sample of Confirmation Dialog

Figure B.7: Sample of Restriction Using Communication Mediation Method

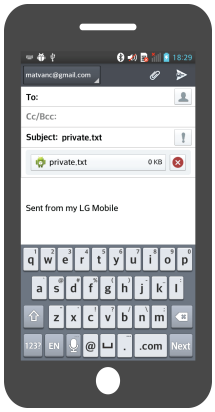


Figure B.8: Sample of Restriction Using File Protection Method

Appendix C

DVD Content

This technical report is accompanied by 3 DVD Dual Layer (8.5 GB). It contains the compressed VirtualBox®Virtual Machine with Xubuntu® Linux OS divided into three parts. The main content – source files of application and TEX system, test files and additional presentation material is located on `DVD 1`. The application source files include – *PE Private Files* configuration Android application, original and pre-prepared repackaged *OI File Manager* and *Ted (Text Editor)* Android applications, as well as original and modificated Aurasium open-source project with the introduced monitoring code handling the tainting and restriction mechanism, and repackaging script. A presentation material contains material from Excel@FIT conference – a poster, pitch slide and insight image – in addition to the slides for presentation. There are also included designed logos, icons and other graphics invented as part of this thesis.

DVD 1

/tex TEX Source Files

/src Application Source Files

/test Application Test Files

/pres Presentation Material for Diploma Thesis

/vm VirtualBox®Virtual Machine (Part 1)

/Diploma Thesis.pdf Diploma Thesis in Electronic Format

/Diploma Thesis – Public.pdf Diploma Thesis in Electronic Format (Public Version)

DVD 2

/vm VirtualBox®Virtual Machine (Part 2)

DVD 3

/vm VirtualBox®Virtual Machine (Part 3)