BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

PRIVACY PROTECTION ON MOBILE DEVICES

OCHRANA CITLIVÝCH INFORMACÍ NA MOBILNÍCH ZAŘÍZENÍCH

PHD THESIS
DISERTAČNÍ PRÁCE

AUTHOR — **Ing. LUKÁŠ ARON**
AUTOR PRÁCE

SUPERVISOR — **Doc. Dr. Ing. PETR HANÁČEK**
ŠKOLITEL

BRNO 2017

Abstract

This thesis analyses privacy protection on mobile devices and presents the method for protecting these data against information leakage. The security is focused on using the mobile device for personal purposes and also for the working environment. The concept of the design solution is implemented in the form of prototype. Model of implementation is verified with the model of required behavior. The thesis also consists of experiments with prototype and verification experiments on defined models.

Abstrakt

Tato práce analyzuje ochranu citlivých dat na mobilních zařízeních a představuje metodu pro ochranu těchto dat před možností úniku informaci ze zařízení. Ochrana se zaměřuje na využívání zařízení, jak pro osobní účely, tak i v pracovním prostředí. Koncept navrženého řešení je implementován ve formě prototypu. Model implementace je verifikován s modelem požadovaného chování. Součástí práce jsou experimenty s prototypem a experimenty zaměřené na verifikaci mezi danými modely.

Keywords

Mobile Device, Privacy Protection, BYOD, Access Rights Model, Android, Verification, Uppaal

Klíčová slova

Mobilní zařízení, Ochrana soukromí, BYOD, Model přístupových práv, Android, Verifikace, Uppaal

Reference

ARON, Lukáš. *Privacy Protection on Mobile Devices.* Brno, 2017. PhD thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Hanáček Petr.

Privacy Protection on Mobile Devices

Declaration

Prohlašuji, že jsem tuto dizertační práci vypracoval samostatně pod vedením pana Doc. Dr. Ing. Petra Hanáčka

Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Lukáš Aron
November 5, 2017

Acknowledgements

Rád bych poděkoval svému školiteli doc. Dr. Ing. Petru Hanáčkovi za odborné vedení a spolupráci při výzkumu. Dále bych rád poděkoval všem, kteří mi poskytli odborné rady a cenné podněty. Děkuji také své rodině a přítelkyni za trpělivost a podporu.

Contents

Chapter 1

Introduction

Nowadays, mobile devices are an essential part of our everyday lives since they enable us to access a large variety of services. In recent years, the availability of these mobile services has significantly increased due to the different form of connectivity provided by mobile devices. In the same trend, the number and typologies of vulnerabilities exploiting these services and communication channels have increased as well.

Therefore, mobile devices may now represent an ideal target for hackers.

As the number of vulnerabilities and, hence, of attacks increase, there has been a corresponding rise of security solutions proposed by researchers. Related to the popularity of mobile devices and also many services that these devices provide the need of protection of user's data are required from the mobile device operating systems [183]. The mobile device is usually controlled by the person who is the owner of the current device.

This person uses its device as a personal device, which means that the content of the mobile device is not explicitly restricted against data leakage, even the private data could be sensitive. There are many mechanisms for protecting the user against data leakage, these tools are built into operating systems or can be additionally installed [98]. The user can work with the device in two modes - public and protected. During working with a public method, the user does not have access to sensitive information and also protected applications. The switch between these modes is usually at the boot time of the device. In some situations, this switching mechanism can be considered as a disadvantage.

There is need of users to use their devices in the working environment, and there exist many solutions to provide this approach, and also these solutions try to prevent the data against its leakage. For instance, the bring your own device (BYOD) approach [29] or mobile device management (MDM) approach [61]. These approaches usually have administration access to the device and can control the device remotely.

These principles can installed and then administration routine can be done without user's knowledge. Th whole device is under the administration of the working institution. The advantage of this principle is that the user work with the device in one mode, which is usually protected and all data are considered as protected. However, the institution can control the sensitive data it also has access to personal data of the owner of the device.

1.1 Goals

The goal of this thesis is to investigate privacy protection on mobile devices and current solutions provided by the manufacturer of these devices, operating system vendors, third-

party institutions, and researchers. Moreover, the investigation covers the software vendors primarily creators of operating systems and their approach to protecting users against data leakage. This thesis concerned with mobile device operating systems, their protection and possibility of enhanced security in the area of information leakage. End users usually use these devices and operating systems for providing a connection with other users, sending messages, using the internet, playing music or movies, taking photos, and work with it. The focus here is on combining mobile device as a personal device and also using it as a work device. This concept covers privacy protection against data leakage. A user is usually an individual unique person, and mostly the only owner of the invention and the possibility of using the personal device at work has its benefits.

Overview of research in this area, possible vulnerabilities and protection against data leakage can also be considered as one of the goals. Besides, there is presented the topic with the threats that are well known and are still valid. In order to understand what issues and protection mechanism are available during these days, the investigation should consider the technical background of a modern operating system with the aim on mobile devices. Moreover, it should also cover approaches to protection, security aspects and the whole protection mechanism implemented in these operating systems.

The investigation of vulnerabilities, properties leading to protection against these vulnerabilities and also defining the novel concept, formal model describing the required behavior, possible prototype implementation as proof of concept and even the verification of application against behavior needed is the primary goal. The focus of this thesis is the definition of the novel approach of using a mobile device as a personal device and also as a work device. The additional value should be that the user has a power of deciding which data is considered as personal and which information is private. Moreover, the institution in which the device is used does not have the control over the device and does not have access to personal data of an owner of the mobile device. This approach determines that the thesis is focused on levels of security with properties such as confidentiality, integrity, and protection.

The formal model of required behavior can be created after the research of current state of protection in the area of mobile devices, exploration of technical details of approaches to this field of security provided by operating systems and also a definition of the behavior. Model of implementation that can be created with the implementation of the prototype should satisfy required properties. To prove that these features are fulfilled the verification of these models are presented.

The verification can be presented on mathematical bases, or existing verification tool can be used. To use existing verification tool, the small research in this area is needed, because there are plenty of verification tools that are general for any verification, or specifically focused on any field. To provide the results from that tool the convenient one should be presented.

1.2 Contribution

The contribution of this thesis is in two things. The first one is the definition of general protection model (model of required behavior), which should consider the need of working with available information on personal basis and another type of information that is marked as private. The personal information or data is marked as public and private data could be then marked as private. The contribution of this work is to provide a possible approach of using the one mobile device for both use cases (personal device and working device)

without any restriction, or without rebooting the machine into a secured mode. Moreover, the security in case of protection against data leakage should be possible without any modification of the operating system. Note that the applications should provide the same functionality as before.

The differences should be apparently visible during protection mode only. The general protection models should be defined formally, with the operating system independence. Therefore this system can be used on any platform even outside mobile world. In contrast, the model of implementation can be operating system dependent related to the creation of this model is based on the implementation of the prototype. This prototype needs to be implemented for the specific operating system with its specifics on security aspects, such as installation applications, access rights, data protection, and other kinds of implementation techniques. The contribution should be confirmed during the verification phase of implementation model with the model of required behavior. Also, experiments of the verification process discuss the possible results, and implementation experiments summarize findings related to the prototype.

Chapter 2 provides introduction into privacy protection on mobile devices. This chapter also describes the complete overview of enhancements of protection user's private information. Additionally, there is the description of security threats targeted at the mobile world. These threats are split into few categories in which the privacy is protected in a different approach. In the second part of the chapter, the protection against discussed security threats are presented.

The chapter 3 discuss protection principles used in modern operating systems. These principles are not limited to mobile operating systems only, but they have also used on desktop or server machines. The first part of the chapter is focused on fundamental properties of a security system with the discussion about confidentiality, integrity, and availability. Access control models are defined formally, and their implementation covers the second part of that chapter.

Mobile operating system architecture is presented in the chapter 4. During this chapter, the general privacy protection defined in the chapter 3 is presented as the implementation of one of the most used operating system for mobile devices. The user data protection is presented through all levels of an operating system with the detail description of the implementation solution. This chapter can be considered as technical background for the implementation of the prototype.

The main idea of this thesis is presented in the chapter 5 that is focused on the definition of access rights model. This chapter discusses the related work in this area of protection user's data and their approaches. The contribution of this thesis starts with this chapter. The most important part of this chapter is the model of required behavior, which is defined formally. This formal model can be then used as the model used in the verification process of the implementation.

Next chapter 6 presents the technical details of the operating system, which have an impact on the required behavior. The system design is described together with the prototype implementation which should satisfy the requirement presented in the previous chapter. The inseparable part of this chapter is the description of a framework that helps define the correct security layer. The second part of this chapter describes the formal model of implementation, that is the simplified model used in a verification process. The model is presented in a similar format as the model of required behavior.

A verification process is described in the chapter 7, in which model of required behavior and model of implementation are verified with the aim of security properties. This chapter

introduces the verification tools that can be used for checking two models. The specification of models in the format of the tool is presented, and other necessary parts for the verification process are defined.

Chapter 8 shows the verification experiments on already defined models. These experiments are considered as an example of possible verification processes between two models. They are focused on general verification of available file operations, and their classification according to a definition. These experiments are evaluated, and the results are discussed at the final stage of each experiment section.

Implementation experiments are presented in the chapter 9, which covers the experiments on a real mobile device and operating system simulator. The implementation is tested via randomly chosen and freely available applications, which are installed and then tested with the implementation solution. The results of these experiments are then collection and evaluated. The main focused is considered on the data leakage, but also on properties such as performance measurement.

The last chapter 10 of this thesis is dedicated to a conclusion.

Chapter 2

Privacy Protection

This chapter covers the overview of privacy protection on mobile devices. Moreover, this chapter describes the privacy protection enhancement designed by other researchers, security specialists, and developers among mobile devices. Additionally, there is the summary of the proposals, and other research works in recent years in this area. A primary aim of this part is related to the leakage of data. Thus, the aim can be seen in two categories - privacy protection enhancement and privacy leakage detection [171]. An approach aimed to privacy protection enhancement can be implemented in the system layer and also in the application layer. The system-level enhancement performs the deficiency of privacy protection mechanism. For instance, coarse granularity access control which allows sensitive data to be leaked out of the device through the implicit data flow. Moreover, relatively sophisticated privacy leakage detection techniques such as taint analysis and flow control analysis many kinds of research have applied machine learning to detect the information leakage.

There are papers related to security survey on mobile devices such as [129, 140, 221]. They are focused on threats, vulnerabilities, and related solutions in the mobile world. A set of tools have been developed by Enck et al. [73] to handle the security issues of applications available through application market. Major security fields have been analyzed, characterized and categorized by Tan et al. [213].

Privacy protection on mobile devices can be categorized in many different aspects. For instance, these selections can be based on threats, platform architecture levels, hardware and software security accesses, operating systems and its models. The first part of this chapter is going to cover the selection privacy based on privacy security threats related to sensitive data access, and then it follows with the leakage information possibilities.

2.1 Privacy Security Threats

As was discussed earlier, the mobile device is an important part of human life, and therefore users store private and sensitive data on it. Due to the mobile operating platform, the amount of malicious applications on this platform is increasing year by year. It is not only the issue of one platform but the whole mobile world. In these days the mobile device with the operating system is just tool to handle usual routines such as phone calls, sms/mms messaging, internet access, camera, music/movie player, and other available functionalities. However, this list of features is not complete. Users want more than limited functionality provided by the manufacturer. This need is the reason for the available ap-

plications provided by the third-party developers. Additionally, these applications enhance the functionality of the device. In contrast, these applications need access to the device hardware through drivers and also access to the system information and sometimes access to the private or sensitive information, such as location, contact list, emails, messages and many other data available on the device. Information or data available on the device can be categorized into four groups, the main idea has been proposed by Midi et al. [160]. These four groups are device resources, user data, system information and application data.

Device Resources

Current modern mobile devices come to the market with many hardware units to provide the specific feature to the user. For instance, one of these units can be near field communication (NFC) [127], global position system (GPS) [243], camera and other sensors which enable applications to accomplish complex functions and services, such as phone navigation [152]. In contrast to desirable properties of all sensors and hardware units, there is also a possible risk for the user. For example, the NFC hardware unit is used in these days for payment access control and ticketing (micropayment) [151]. According to NFC, Haselsteiner et al. [102] found possible NFC threats, such as eavesdropping [149], data corruption, data modification, data insertion, man-in-the-middle attack [65], these attacks or threats are available because the principle of this technology is based on wireless communication.

Another manner of sensitive information detection related to resources available on a modern mobile device has been found by Dey et al. [67]. They found that hardware imperfection during fabrication process makes each accelerometer sensor chip unique. The uniqueness is in the response that each chip response is different to the same motion stimulation. This knowledge makes the device to easily track a user over space and time because this motion stimulus creates a unique fingerprint of the user.

User Data

User data is produced when any user uses the device. Therefore, a user is using the basic services of a mobile device, including sms/mms messages, contact list, phone records and other mobile features. Felt et al. [77] found that the most common malicious behavior is the stealing of personal information of users. Additionally, the most instant messaging software request access to the contact list and use the address book to recommend friends to users. Although it becomes convenient, attackers can utilize this method to obtain the user's privacy information automatically.

System Information

System information consists of various sources of the current mobile device. These information could be international mobile equipment identity (IMEI) [125], international mobile subscriber identity (IMSI)[189], phone number, Wi-Fi media access control (MAC) address [59], etc. Some information can uniquely identify a mobile device, namely, identify a unique user. According to permissions of the operating system applications installed on the device can access this information. Related to the research of Achara et al. [1], they found that permission which allows access to the state of wireless connection can not only identify the user with MAC address but can also obtain their coarse-grained location information without requesting the permission for location.

Application Data

Applications create various data related to the user's interest and cookies in the case of internet browser. Although on every modern operating system aimed to mobile devices is application sandbox mechanism available to ensure the isolation between applications, there are still some means for malware [37] to gather and analyze these data. Recently, more applications use browser kernel to show the hypertext markup language (HTML) [166] content within them. However, the trend in the development of mobile applications raises in the idea of developing an application for all platforms. The main solution to this idea is to use the browser kernel to handle view part of the application in HTML, and the logic is maintained by JavaScript [79]. Approaches to developing application through more than one operating system are out of the scope of this thesis, but some additional information can be found in [9, 85, 150].

One feature of this approach is that provides a way for JavaScript in a browser kernel to invoke application code when the application enables the kernel. This principle allows the web page to access functionality and also data exposed by the application, which undoubtedly increases application's attack. This vulnerability has been published by Chin et al. [55] on webview [39].

2.2 Privacy of Communication Channels

Related to a weakness of privacy communication channels, researchers try to handle this security breaches and propose the tools and mechanisms to protect a user against privacy leakage. In this section are two types of leakage. One is related to permission escalation, which is aimed at some operating system privacy issue. The second one is an informal overview of possible collusion attack.

Permission Escalation

This security issue is related mainly to an operating system and is described as an application with fewer permissions (a non-privileged caller) is not restricted to access components of a more privileged application (a privileged callee) [62].

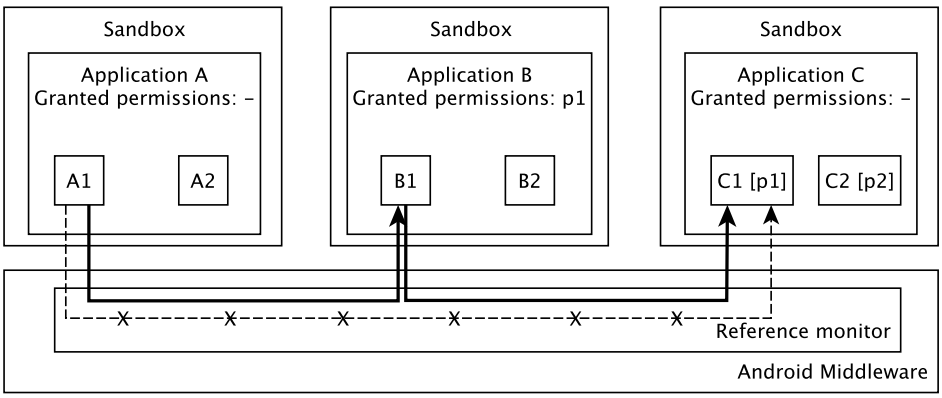

Figure 2.1: Component-based permission escalation attack [62]

In other words, operating system security architecture does not ensure that a caller is assigned at least the same permissions as a callee. Figure 2.1 shows the situation in which privilege escalation attack becomes possible. Applications A, B and C are assumed to run on an operating system, each of them is isolated in its sandbox. Application A has no granted permissions and consists of components $A1$ and $A2$. Application B is granted a permission $p1$ and consists of components $B1$ and $B2$. Permission labels protect neither $B1$ nor $B2$ and thus can be accessed by any application. Both, $B1$ and $B2$ can access components of external applications protected with the permission label $p1$ since in general, all application components inherit permissions granted to their application. Application C has no permissions granted, it consists of components $C1$ and $C2$. Components $C1$ and $C2$ are protected by permission labels $p1$ and $p2$, respectively, that means that component $C1$ can be accessed only by components of applications which possess $p1$, while component $C2$ is accessible by components of applications granted permission $p2$.

As can be seen in figure 2.1, component $A1$ is not able to access $C1$ component, since $p1$ permission is not granted to the application A. Nevertheless, data from component $A1$ can reach component $C1$ indirectly, via the $B1$ component. Indeed, component $B1$ can be accessed by component $A1$ since component $B1$ is not protected by any permission label. In turn, component $B1$ is able to access $C1$ component since the application B and consequently all its components are granted $p1$ permission [62].

Collusion Attack

Attackers accomplish malicious behavior by colluding applications and therefore indirectly escalate their permissions. Collusion attack can escape those detection technology designed for a single application. Collusion attack has been revealed by the Marforio et al. [153]. They implemented and analyzed much covert and overt communication channel that enables applications to collude.

2.3 Privacy Protection Mechanisms

According to the weakness of privacy protection mechanisms on mobile devices, researchers try to improve the system privacy at different levels. These levels are described in the following sections.

Privacy Protect Enhancement

Enhancement of privacy can be one on system layer, which is related to operating system and also on the application layer. The application layer is based on third-party developers and also on execution platform which is the operating system. Lets firstly describe the system layer and after that the application layer security enhancements.

System Layer

The operating system provides a set of permission to limit applications access to sensitive resources [7, 135]. But the concept of *all-or-nothing* feature make it weak [7]. There is an approach which provides appropriate authorization for access system resources. Design and implementation of this approach are based on graph-theoretical algorithm [89]. Moreover, Shen et al. [201] proposed flow permission mechanism provide users additional context on how the applications leverage the standard permissions and resources.

From another point of view of security - isolation. The isolation technique has been used in Solaris operating system [2], Linux based operating system [147], etc. Applications or processes can not interact with each other in an isolated environment because protect application data. Better isolation is achieved with the solution called AirBag [227]. It is a lightweight operating-system level virtualization approach which is designed to isolate and prevent malware from infecting systems or stealthily leaking private information. However, numerous system events are isolated at AirBag boundary as result of confining the untrusted application to communicate with other legitimate applications and services running on the native run-time, which will affect the certain functionality of untrusted applications. This solution is focused on isolating privacy information on data level or process level.

A similar approach has been proposed by Lange et al. [132] which is called Crossover. Crossover is based on L4Android [133] and provides a framework which defines the requirements and properties of a secure and usable user interface to manage several different operating system environments on one device.

To provide a safe business environment based on BYOD principle [163], Wang et al. [225] proposed an enterprise-level security policy enforcement mechanism called DeepDroid. It is a fine-grained system which modifies the system services responsible for access rights. Enterprise administrators can dynamically enforce fine-grained system services and resource access control policy. Although, various works of privacy protection enhancement, such as TrustDroid [245], using a different security policy, it is approximately the same approach where they hook and how they enforce a policy.

For mobile operating system exists at least two proposals of policy enforcement published by Backes et al. [27] and Heuser et al. [105] respectively proposed the Android security framework (ASF) and Android security modules (ASM) framework, providing a programmable interface to develop another novel protection mechanisms. The main difference between ASF and ASM is that the ASF is deployed in the bottom of existing ASF, allowing third-party developers to supplement or replace the existing platform security mechanism, while ASF enhances the system security and privacy through applications.

To be sure, that data is unable to use, when other access control mechanisms such as authentication or file access are compromised, data encryption technology can guarantee that data is not possible to read. Implementation of a practical system of encryption on mobile devices is published by Yu et al. [241]. This proposal is called MobiHydra, and the implementation is based on plausibly deniable encryption [46], featuring multilevel deniability on mobile devices. A user can choose to store sensitive data at different deniability levels and can hide data without rebooting in this system. Besides, to focus on the entire file system encryption, there is another proposal called MobiPluto [51] which is based on a principle of denying the existence of sensitive data stored on the mobile device.

Another type of security protection is security authentication. An Authentication is presented on a mobile device on many levels, but the first one is the key to open the device for usage. This principle is usually done by personal identification number code, pattern lock, face recognition, or any other features which could uniquely identify the user of the device. Chiang et al. [53] proposed a new multiple-layer graphical password scheme, which allows users to draw their passwords across multiple layers through the „warp-cells.“ This approach avoids unlocking the device by brute force. A different approach for a low participation of authentication mechanism proposed Li et al. [138]. This proposal uses a classifier to learn the owner's finger movement pattern. It can continuously re-authenticates the current user without interrupting user-smart-phone interactions.

Application Layer

The current privacy protection enhancement research on application layer mainly covers two aspects: recommendation to the users to fewer risk applications according to their risk level, and taking a positive response to defense mechanisms, such as monitoring application's behavior while they are running.

The application risk assessment introduces machine learning methods to privacy protection. For instance, Peng et al. [174] use probabilistic generative models for risk scoring schemes, and identify several models ranging from the simple Naive Bayes models [184] to advanced hierarchical mixture models. Moreover, Zhu et al. [247] developed a recommendation system that considers both the application's popularity and the privacy threat. This solution is based on the work of [174]. Different from using permission to assess risk an automated framework called RISKMON was presented by [116]. This framework scores the risk based on user's coarse expectations and application's behavior. Although, machine learning needs a huge amount of training data and therefore the data model which contains information which facilitated risk analysis was created by [212].

Monitoring run-time process is helpful for users to understand the privacy data flow and detect the malicious behavior. For instance, the FireDroid [186] monitoring tool serves as a monitor process which controls the execution of native codes and prevents privacy leakages. It is working transparently to a user. Moreover, intrusion detection system based on a host can report and interrupt malicious activity in real-time. This mechanism improves the defense of the host system. To prevent intrusion by performing run-time policy enforcement on system-level, the *Patronous* - security architecture system proposed by Sun et al. [211].

Privacy Leakage Detection

Private leakage detection research is focusing primarily to taint analysis, control flow analysis, and virtualization. Also, some researchers introduce machine learning principles to privacy leakage detection.

Taint analysis includes static and dynamic taint parts. It should taint the sensitive information firstly, and then analyze the data flow through taint tracking or alias analysis algorithm. A novel static taint analysis system called FlowDroid [23]. This system is context, flow and objects sensitive while precisely modeling life-cycle, it can adequately handle callbacks invoked by the framework. Unfortunately, it only enforces taint analysis between single components. Another proposal, which is based on TaintDroid [72] is NDroid [178] and it performs dynamic taint analysis. This system is designed for checking information flows through Java native interface (JNI) [91]. NDroid can work together with TaintDroid to track information flows from selected sources to specified sinks in applications.

Applications which are framework-based and event-driven which lead to traditional control flow analysis are no longer adaptive. In order to tackle this issue, the new program representation was proposed, and it is named callback control-flow graph (CCFG) [236]. An algorithm presented for CCFG is based on construction through context-sensitive control-flow analysis of callbacks. Moreover, automated privacy leakage detection system called AAPL [148] based on the multiple special static analysis techniques including flow identification and joint flow tracking. Additionally, AAPL uses peer voting to filter out legitimate privacy disclosures purifying the detection results. The cons of the AAPL are an impossibility to detect disclosures caused by Java reflection [80], code encryption, or dynamical code loading.

The most mobile applications are written in a programming language which is not the same as the programming language used to develop a kernel of the operating system. For instance, Android applications are usually developed using Java programming language [92], while the underlying kernel is implemented by C programming language [121]. The similar approach is with the operating system iOS [110]. The kernel of iOS is also implemented by C programming language and applications are developed in Swift programming language [226]. Virtualization technologies are necessary for this area and provide a virtual execution environment for dynamic detecting privacy leakage behavior while preventing other applications from being infected. Compared with static analysis, it has a higher precision as the behavior is detected at run-time. For instance, DroidScope [235] is one of many malware analysis detection platform. It is based on top of quick emulator (QEMU) [31] (multiple-host emulator for multiple-targets), and it can reconstruct the operating system-level and also virtualization-level semantic views. Apart from this, DroidScope provides a set of APIs to help researchers implement custom analysis plugins.

Installation packages of the available mobile applications contain compressed much information, that can be helpful for researchers. For instance, requested permissions, graphical user interface, and compiled code. Additionally, the data flows gained by the traditional methods such as methods described in this section - taint analysis and control flow analysis can be trained as features for machine learning. This approach was proposed by Tripp et al. [215] based on the Bayesian notion of statistical classification. These classifications have conditioned the with judgment whether a release point is legitimate on the evidence arising at that point. Another similar solution called MUDFLOW [24] can detect abnormal flows in possibly malicious applications through learning abnormal and normal flows of sensitive information from trusted applications.

2.4 Summary

The introduction into the privacy protection on mobile devices has been covered in this chapter. There were also presented some other work of researchers, security specialist and developers with a focus on security threats and vulnerabilities. These projects are trying to solve the security breaches, malware protection, vulnerability protection or just move the security on mobile devices into a more secure sphere. The aim of the thesis is the protection against data leakage, and these presented projects are related to this topics. Specifically, the second part of this chapter discussed the protection and also detection of privacy leakage.

In order to understand the theoretical background around this topic, the next chapter discusses the protection mechanism used primarily on an operating system level, but these principles are also used on the higher levels of a software stack. These mechanism related to the protection are not limited to mobile operating systems.

Chapter 3

Operating System Protection

This chapter introduces the operating system protection mechanism widely used and distributed with a responsibility to protect users, applications against security breaches, malware infection and also against information leakage [128]. The operating system itself is a software that communicates directly with hardware or hardware drivers. The main principle is to provide hardware features to users of this system. The security on the operating system level is mainly focused on protection mechanism and its control access.

Protection mechanisms control access to a system by limiting the types of file access permitted to users. Also, protection has to ensure that only processes that have gained proper authorization from the operating system can operate on memory segments, the processor, and other computer resources.

Protection is provided by a mechanism that controls the access to applications, processes, or users to the resources defined by a computer system. These mechanisms have to provide a means for specifying the controls to be imposed together with a means of enforcing them.

Security ensures the authentication [83] of system users to protect the integrity of the information stored in the system, as well as the physical resources of the computer system. The security system prevents unauthorized access, malicious destruction or alteration of data, and accidental introduction to inconsistency.

As computer systems have become more sophisticated and pervasive in their applications, the need to protect their integrity has also grown. Protection was initially conceived as an adjunct to multi-programming operating systems, so those untrustworthy users might safely share a standard logical to, such as a directory of files, or share a joint physical concerning, such as memory. Modern protection concepts have evolved to increase the reliability of any complex system that makes use of shared resources.

The most obvious reason to have protection mechanisms is the need to prevent the mischievous, intentional violation of an access restriction by a user. Of more general importance, however, is the need to ensure that each application component active in a system uses system resources only in manners consistent with stated policies. The requirement is an absolute one for a reliable system.

Protection can improve reliability by detecting latent errors at the interfaces between component subsystems. Early detection of interface errors can often prevent contamination of a healthy subsystem by a malfunctioning subsystem. Moreover, an unprotected resource cannot defend against use by an unauthorized or incompetent user. A protection-oriented system provides means to distinguish between authorized and unauthorized usage.

The role of protection in a computer system is to provide a mechanism for the enforcement of the policies governing resource use. These procedures can be established in a variety of ways. Some are fixed in the design of the system, while others are formulated for the management of the system. Still, others are defined by the individual users to protect their files and applications. A protection system has to have the flexibility to enforce a variety policies.

Policies for resource use may vary by application, and they may change over time. For these reasons, protection is no longer solely of the designer of an operating system. The application programmer needs to use protection mechanisms as well, to guard resources created and supported by an application subsystem against misuse.

3.1 Fundamental Properties of Secure System

The basis of computer security issues is made up of three fundamental properties. They describe the accessibility of the system, the correctness of any manipulation of any object on the system and to what extent information considered sensitive is kept secret. The properties are called *availability, integrity* and *confidentiality* and they will be described in this section. The definitions are from these sources [11, 42]. Each section will begin with a discussion and proceed with the definition of the current term.

Confidentiality

To keep data and its existence secret is a challenge that many organizations put a significant amount of time and money. To keep data confidential is a major concern too, for instance, intelligence agencies and the military, where information is often made available to personnel on a *need to know* basis. Cryptography is an important part of the implementation of private systems. A sensitive example of the use of confidentiality is the medical records that are stored in medical databases. In these databases, information about one's emotional health inherited diseases, and more is stored. Most people consider this information to be very private and do not want anybody but perhaps their doctor to know about it.

Definition 3.1.1. Let X be a set of entities and let I be some information. Then I has the property of confidentiality with respect to X if no member of X can obtain information about X.

Integrity

In most commercial environments, the integrity of information is more important than to protect it from unauthorized access, although that too is an important issue. For instance the importance of integrity in a bank's transaction records or the contents of a gas station's selling record. There exist two main categories of *integrity* mechanisms:

- detective integrity mechanisms
- preventive integrity mechanisms

Detective integrity mechanisms are used to detect any unauthorized modification to information. The mechanism may give a detailed report under which circumstances the information's integrity was affected: by whom and what part of the information that was

affected, or it may just report that the data has been changed and mark the data as no longer trustworthy.

Preventive integrity mechanisms try to maintain the integrity of any information by blocking any unauthorized attempts to modify it. This mechanism also includes the case when a user that has been authorized to modify some information in a certain way tries to alter it in an unauthorized manner.

Definition 3.1.2. Let X be a set of entities and let I be some information. Then I has the property of integrity with respect to X if all members of X trust I

Availability

One of the most basic aspects of a system is its availability. If a subject is unable to utilize the services provided, the service may just as well not exist. Any interruption in the availability of the system's parts will make the availability of the entire system to fail.

Definition 3.1.3. Let X be a set of entities and let I be a resource. Then I has the property of availability with respect to X if all members of X can access I.

3.2 Trusted Computing Base

In the security, the world is using the term trusted systems rather than secure systems. These systems that have formally stated security requirements and reach these requirements. At the core of every trusted system is a minimal trusted computing base (TCB) [176, 185] which is defined by hardware and software necessary for enforcing all the security rules. The ability of a trusted computing base to enforce a security policy correctly depends foremost on the integrity, correctness, and protection of the mechanisms implementing the elements of the TCB itself. Similarly, a network trusted computing base (NTCB) [42] is defined as the totality of protection mechanisms within a network including hardware, firmware, and software, the combination of which is responsible for enforcing a network-wide security policy.

A mechanism is a term used to refer to a specific paradigm, model, or a construct that is used in the implementation of a particular service. A security service enforcing a policy is, therefore, a combination of security mechanisms. Trust in a TCB means the components and mechanisms are implementing the enforcement of controls dictated by a security policy behave expectedly. The expectation here is that the TCB should not subvert the policy that it is designed to enforce. Essential to the element of trust in the TCB is its correctness and overall system integrity.

The general method of defining the boundaries of a TCB is that any software, firmware, or a hardware component that can subvert a security policy is considered to be part of an applicable TCB or NTCB. Breaching a TCB is usually accomplished by carrying an attack that the designer of the TCB had not anticipated. Building an ideal TCB, therefore, requires exhausting all possible attacks. While it may seem that the elements of network TCB are scattered and disjoint, in practice trust is a continuous concept throughout that follows the information flow. Applicable trust properties should remain invariant when information is residing on a storage system, within a thread of execution, during an exchange of data across address spaces, or while in transmission over a network.

Since the TCB is working to specification, the system security cannot be compromised even when something else is wrong. An essential part of the TCB is the reference monitor

[10]. A reference monitor is the TCB component of a computing system that mediates every access of a subject to a resource following a security policy that governs such access. The policy may be implemented in the form of rules and attributes associated with a registry of subjects and a registry of objects. The rules can be static access rights (permissions), roles, or dynamically deduced rights.

Figure 3.1 illustrates the concept of an access-control reference monitor. In addition to the mediation of access, a reference monitor should not be bypassed at all times, should support isolation of the security services from untrusted processes, maintain system integrity, and prevent from tampering by users or system processes.

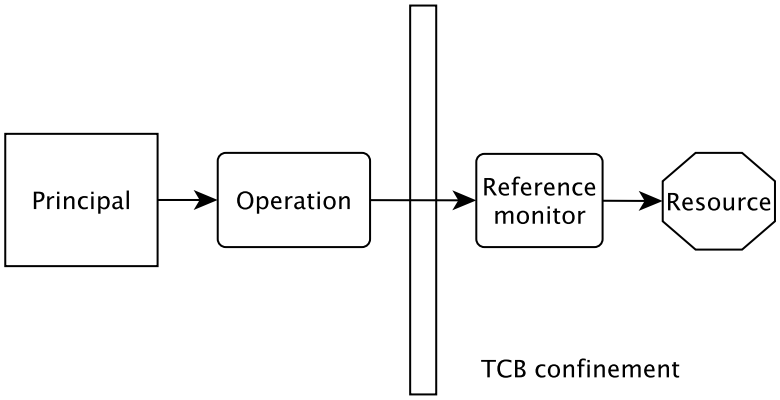

Figure 3.1: A reference monitor concept of access control citeanderson1972computer

The reference monitor footprint should be kept small enough to be susceptible to rigorous verification methods. The gate-keeper approach of the reference monitor makes it an ideal component for the generation of audit trails reflecting access attempts to the resources within its confines.

3.3 Users, Principals, Subjects, and Objects

The term *user* in computing has been traditionally equated with a human being. Its use conveys a unique association between a computing system and an entity that can be a human being or some programmable agent. User information is encapsulated in an account, sometimes referred to as a profile. A user account contains information about authentication as well as authorization credentials and may contain a set of attributes describing the user (such as a name, a serial number, an organization name, and so forth). Each user account is associated with an identifier that must be unique in the naming space of the underlying computing system.

While a user represents an entity external to a computing system, a *principal* refers to an entity's internal representation of a computing system. Each user may have several principals associated with it. Each principal, on the other hand, is associated with one user only. The principal construct defines the run-time association between a computing task and a particular user and encapsulates a subset of the entitlements of that user. The scope of entitlement is dependent on the application to which the user is signed.

A *subject* is the term used to identify a running process, a program in execution. Each subject assumes the identity and the privileges of a single principal. A principal may launch

several processes within a single login session and thus will be associated with multiple subjects, each of which inherits the identity of the login session. Figure 3.2 illustrates the relationships between a user, principal, and a subject, defined by [35].

An *object* refers to a passive entity (i.e., one that is an information receptacle such as a file, or a record in a database). An object, however, may indicate an active device from the system's resource pool (such as a network printer, or further can be a programmable service that is managed as a resource).

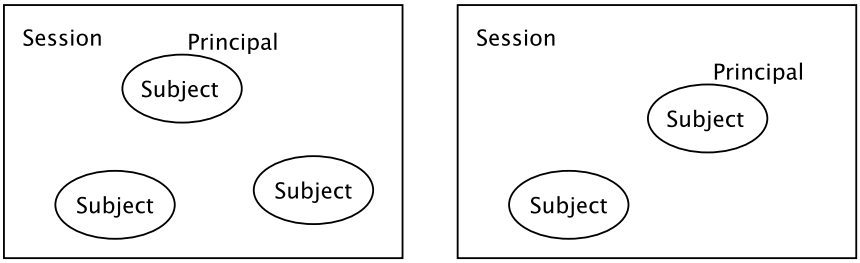

Figure 3.2: Relation between a user (primary principal), a principal, and a subject [35]

It is worth noting that in many cases it merely encounter the basic scenario in the relationships among a user, principal, and subject where the user, the principal, and the subject are all the same. In the security literature, the term principal is used to mean an active entity that is capable of causing information to be retrieved, changed or flown between controlled objects of a computing environment.

3.4 Access Control

An important requirement of any information management system is to protect data and resources against unauthorized disclosure (secrecy) and unauthorized or improper modifications (integrity), while at the same time ensuring their availability to legitimate users (no denial-of-service). Enforcing protection, therefore, requires that every access to a system and its resources be controlled and that all and only authorized accesses can take place. This process goes under the name of access control. The development of an access control system requires the definition of the regulations according to which access is to be controlled and their implementation as functions executable by a computer system. The development process is usually carried out with a multi-phase approach based on the following concepts:

- **Security policy** - it provides the high-level rules according to which access control have to be regulated.
- **Security model** - it defines a formal representation of the access control security policy and its working. The formalization allows the proof of properties on the security provided by the access control system being designed.
- **Security mechanism** - it determines the low-level functionality (defined in software or hardware) that implements the controls imposed by the policy and formally stated in the model.

The three concepts above correspond to a conceptual separation between different levels of abstraction of the design and provides the traditional advantages of multi-phase software development. In particular, the separation between policies and mechanisms introduces an independence between protection requirements to be enforced on the one side, and mechanisms are enforcing them on the other. It is then possible to discuss protection requirements independently of their implementation, and also compare various access control policies as well as different mechanisms that enforce the same policy, and design mechanisms able to enforce multiple policies.

This latter aspect is particularly important: when a mechanism is tied to a specific policy, a change in the policy would require changing the whole access control system, mechanisms able to enforce multiple policies avoid this drawback. The formalization phase between the policy definition and its implementation as a mechanism allows the definition of a formal model representing the policy and its working, making it possible to define and prove security properties that systems enforcing the model will enjoy. Therefore, by proving that the model is secure and that the mechanism correctly implements the model. The implementation of an exact mechanism is far from being trivial and is complicated by the need to cope with possible security weaknesses due to the implementation itself and by the difficulty of mapping the access control primitives to a computer system. The access control mechanism has to work as a reference monitor (see figure 3.1), that is, a trusted component intercepting every request to the system.

Even the definition of access control policies (and their corresponding models) is far from being a trivial process. One of the major difficulty lies in the interpretation of, often complex and sometimes ambiguous, real-world security policies and their translation in well defined and unambiguous rules enforceable by a computer system. Many real-world situations have complex policies, where access decisions depend on the application of different rules coming, for instance, from laws, practices, and organizational regulations. A security policy has to capture all the different regulations to be enforced and, also, has to also consider possible additional threats due to the use of a computer system. Access control policies can be grouped into three main classes:

- **Discretionary acess control (DAC)**[139] policies control access based on the identity of the requester and on access rules stating what requesters are allowed to do (or not allowed to do).
- **Mandatory access control (MAC)**[144] policies control access based on mandated regulations determined by a central authority.
- **Role-based access control (RBAC)**[192] policies control access depending on the roles that users have within the system and on rules stating what accesses are allowed to users in given roles.

Discretionary and role-based policies are usually coupled with (or include) an administrative policy that defines who can specify authorizations/rules governing access control.

Discretionary Access Control

Discretionary policies enforce access control by the identity of the requesters and explicit access rules that establish who can or cannot, execute which actions on which resources. They are called discretionary as users can be given the ability to pass on their privileges to

	File 1	File 2	File 3	Application 1
User 1	own, read, write	read, write		execute
User 2	read		read, write	
User 3		read		execute, read

Table 3.1: Access matrix

other users, where granting and revocation of privileges is regulated by an administrative policy.

Access Matrix Model

The access matrix model proposed by Lampson [131] provides a framework for describing discretionary access control. It is based on protection of resources within the context of operating systems. The model was subsequently formalized by Harrison, Ruzzo, and Ullmann (HRU model) [101], who developed the access control model proposed by Lampson with the goal of analyzing the complexity of determining an access control policy. The original model is called access matrix since the authorization state, meaning the authorizations are holding at a given time in the system, is represented as a matrix. The matrix, therefore, gives an abstract representation of protection systems.

A first step in the development of an access control system is the identification of the *objects* to be protected, the *subjects* that execute activities and request access to *objects*, and the actions that can be executed on the *objects*, and that must be controlled. *Subjects, objects*, and actions may be different in different systems or application contexts. For instance, in the protection of operating systems, *objects* are typically files, directories, or applications. In contrast, in database systems, *objects* can be relations, views, and or stored procedures. It is interesting to note that *subjects* can be themselves objects (this is the case, for instance, of executable code and stored procedures). A *subject* can create additional *subjects* (e.g., children processes) in order to accomplish its task. The creator subject acquires control privileges on the created processes (e.g., to be able to suspend or terminate its children).

In the access matrix model [131], the state of the system is defined by a triple (S, O, A), where S is the set of *subjects*, who can exercise privileges. O is the set of *objects*, on which privileges can be exercised (*subjects* may be considered as *objects*, in which case $S \subseteq O$) and A is the access matrix, where rows correspond to *subjects*, columns correspond to *objects*, and entry $A[s, o]$ reports the privileges of s on o. The type of the *objects* and the actions executable on them depend on the system. By simply providing a framework where authorizations can be specified, the model can accommodate different privileges. For instance, in addition to the traditional read, write, and execute actions, ownership (i.e., property of *objects* by *subjects*), and control can be considered. Table 3.1 depicts an example of access matrix.

Changes to the state of a system are carried out through commands that can execute primitive operations on the authorization state, possibly depending on some conditions. The HRU formalization identified six primitive operations that describe changes to the state of a system. These operations, whose effect on the authorization state are defined in the following definitions [191], correspond to adding and removing a subject, adding and removing an object, and adding and removing a privilege.

Definition 3.4.1. operation **enter** r into $A[s, o]$ [191]:

conditions: $s \in S, o \in O$
new state: $\mathbb{Q} \vdash_{operation} \mathbb{Q}'$
$S' = S$
$O' = O$
$A_1[s, o] = A[s, o] \cup \{r\}$
$A_1[s_i, o_j] = A[s_i, o_j], \forall (s_i, o_j) \neq (s, o)$

Definition 3.4.2. operation **delete** r from $A[s, o]$ [191]:

conditions: $s \in S, o \in O$
new state: $\mathbb{Q} \vdash_{operation} \mathbb{Q}'$
$S' = S$
$O' = O$
$A'[s, o] = A[s, o] \backslash \{r\}$
$A'[s_i, o_j] = A[s_i, o_j], \forall (s_i, o_j) \neq (s, o)$

Definition 3.4.3. operation **create subject** s' [191]:

conditions: $s' \notin S$
new state: $\mathbb{Q} \vdash_{operation} \mathbb{Q}'$
$S' = S \cup \{s'\}$
$O' = O \cup \{s'\}$
$A'[s, o] = A[s, o] \forall s \in S, o \in O$
$A'[s', o] = \emptyset, \forall o \in O$
$A'[s, s'] = \emptyset, \forall s \in S'$

Definition 3.4.4. operation **create object** o' [191]:

conditions: $o' \notin O$
new state: $\mathbb{Q} \vdash_{operation} \mathbb{Q}'$
$S' = S$
$O' = O \cup \{o'\}$
$A'[s, o] = A[s, o], \forall s \in S, o \in O$
$A'[s, o'] = \emptyset, \forall s \in S'$

Definition 3.4.5. operation **destroy subject** s' [191]:

conditions: $s' \in S$
new state: $\mathbb{Q} \vdash_{operation} \mathbb{Q}'$
$S' = S \backslash \{s'\}$
$O' = O \backslash \{s'\}$
$A'[s, o] = A[s, o], \forall s \in S', o \in O'$

Definition 3.4.6. operation **destroy object** o_1 [191]:

conditions: $o' \in O, o' \notin S$

new state: $\mathbb{Q} \vdash_{operation} \mathbb{Q}'$

$S' = S$

$O' = O \backslash \{o_1\}$

$A'[s, o] = A[s, o], \forall s \in S', o \in O'$

Each command has a conditional part and body and has the form defined in the following listing 3.1 with $n > 0, m \geq 0$. There $r', \ldots, r_m$ are actions, $op', \ldots, op_m$ are primitive operations, while $s', \ldots, s_m$ and $o', \ldots, o_m$ are integers between 1 and k. If $m = 0$, the command is without conditional part.

```
command c(x_1, ..., d_k)
  if r_1 in A[x_s1, x_o1] and
     r_2 in A[x_s2, x_o2] and
     ⋮
     r_m in A[x_sm, x_om]
  then op_1
       op_2
       ⋮
       op_n
end.
```

Listing 3.1: Form of the command

For instance, the following command presented in listing 3.2 creates a file and gives the creating subject ownership privilege to it.

```
command CREATE(creator,file)
        create object file
        enter Own into A[creator,file]
end.
```

Listing 3.2: Create command [191]:

Following commands expressed in listing 3.3 allow an owner to grant to others, and revoke from others, a privilege to execute an action on his/her files.

```
command CONFER_a(owner, friend, file)
        if Own in A[owner,file]
        then
                enter a into A[friend,file]
        end.

command REVOKE_a(owner, ex-friend, file)
        if Own in A[owner,file]
```

```
        then
                delete a from A[ex-friend, file]
end.
```

Listing 3.3: Grant permission command [191]:

Note that the variable a from listing 3.3 is not a parameter, but an abbreviation for defining many similar commands, one for each value that a can take (e.g. $CONFER_{read}$, $REVOKE_{write}$). Since commands are not parametric actions, a different command needs to be specified for each action that can be granted and or revoked, depicted in definition 3.4.7.

Definition 3.4.7. Let $\mathbb{Q} \vdash_{op} \mathbb{Q}'$ denote the execution of operation op on state $\mathbb{Q}$, resulting in state $\mathbb{Q}'$. The execution of command $c(a_1, \ldots, a_k)$ on a system state $\mathbb{Q} = (S, O, A)$ causes the *transition* from state $\mathbb{Q}$ to state $\mathbb{Q}'$ such that $\exists \mathbb{Q}_1, \ldots, \mathbb{Q}_n$ for which $\mathbb{Q} \vdash_{op_1^*} \mathbb{Q}_1 \vdash_{op_2^*} \cdots \vdash_{op_n^*} \mathbb{Q}_n = \mathbb{Q}'$, where $op_1^* \ldots op_n^*$ are primitive operations $op_1 \ldots op_n$ in the operational part of the command c, in which actual parameters a_i are substituted for each formal parameters $x_i, i = 1, \ldots, k$.

If the conditional part of the command is not verified, then the command has no effect and $\mathbb{Q} = \mathbb{Q}'$.

Although the HRU model does not include any built-in administrative policies, the possibility of defining commands allows their formulation. Administrative authorizations can be specified by attaching flags to access privileges. For instance, a copy flag, denoted *, attached to a privilege may indicate that the privilege can be transferred to others. Granting of authorizations can then be accomplished by the execution of a command.

Implementation of Access Matrix

Even though the matrix represents a quite good conceptualization of authorization, it is not appropriate for implementation. In a general system, the access matrix will be usually enormous in size and sparse, because most of its cells are with the empty value. Storing the matrix as a two-dimensional array is, therefore, a waste of memory space. There are well known three approaches of implementation:

- **Authorization table**[76] - Access matrix in which the non-empty cells are reported with three columns, corresponding to subjects, actions, and objects, respectively. Each tuple in the table is related to authorization. The authorization table approach is used in database management systems, where authorizations are stored as relational tables of the database.
- **Access control list**[199] - The matrix is stored by columns. Each object is associated with a list indicating, for each subject, the actions that the subject can perform on the object.
- **Capability**[214] - The matrix is stored by rows. Each user has associated a list, called capability list, indicating, for each object, the accesses that the user is allowed to exercise on the object.

Capabilities and access control lists (ACLs) [191, 199] present advantages and disadvantages concerning authorization control and management. In particular, with ACLs, it is

immediate to check the authorizations holding on an object, while retrieving all the authorizations of a subject requires the examination of the ACLs for all the objects. Analogously, with capabilities, it is immediate to determine the privileges of a subject, while retrieving all the accesses executable on an object requires the examination of all the different capabilities. These aspects affect the efficiency of authorization revocation upon deletion of either subjects or objects. In a system supporting capabilities, it is sufficient for a subject to present the appropriate capability to gain access to an object.

It represents an advantage in distributed systems since it permits to avoid repeated authentication of a subject: a user can be authenticated at a host, acquire the appropriate capabilities and present them to obtain accesses at the various servers of the system.

Mandatory Access Control

Mandatory security policies enforce access control by regulations mandated by a central authority. The most common form of mandatory policy is the multilevel security policy, based on the classifications of *subjects* and *objects* in the system. Objects are passive entities storing information. Subjects are active entities that request access to the objects. Note that there is a distinction between *subjects* of the mandatory policy and the *authorization subjects* considered in the discretionary policies. While authorization subjects typically correspond to users (or groups), mandatory policies make a distinction between *users* and *subjects*.

Users are human beings who can access the system, while subjects are processes (i.e., applications in execution) operating on behalf of users. This distinction allows the policy to control the indirect accesses (leakages or modifications) caused by the execution of processes.

Security Classifications

In multilevel mandatory policies [191], an access class is assigned to each object and subject. The access class is one element of a partially ordered set of classes. The partial order is defined by a *dominance* relationship, which it is denoted with $\geq$. While in the most general case, the set of access classes can simply be any set of labels that together with the dominance relationship defined on them form a partially ordered set, most commonly an access class is defined as consisting of two components: a *security level* and a *set of categories*. The security level is an element of a hierarchically ordered set, such as top secret (TS), secret (S), confidential (C), and unclassified (U), where $TS > S > C > U$. The set of categories is a subset of a unordered set, whose elements reflect functional, or competence areas, for instance, military systems, financial systems, and other types of systems.

The dominance relationship $\geq$ is then defined as: an access class c_1 dominates ($\geq$) an access class c_2 if the security level of c_1 is greater than or equal to that of c_2 and the categories of c_1 include those of c_2.

Formally, given a totally ordered set of security levels $\mathcal{L}$, and a set of categories $\mathcal{C}$, the set of access classes is $\mathcal{AC} = \mathcal{L} \times \mathcal{P}(\mathcal{C})$[1], and $\forall c_1 = (\mathcal{L}_1, \mathcal{C}_1), c_2 = (\mathcal{L}_2, \mathcal{C}_2) : c_1 \geq c_2 \iff \mathcal{L}_1 \geq \mathcal{L}_2 \wedge \mathcal{C}_1 \supseteq \mathcal{C}_2$. The two classes c_1 and c_2 such that neither $c_1 \geq c_2$ nor $c_2 \geq c_1$ holds are said to be *incomparable*. The source of definition is journal [191]. It is easy to see that

[1]The symbol $\mathcal{P}$ denotes the powerset, thus $\mathcal{P}(\mathcal{C})$ means the powerset of $\mathcal{C}$.

the dominance relationship so defined on a set of access classes $\mathcal{AC}$ satisfies the following definitions:

Definition 3.4.8. Relationship definitions [191]

- Reflexivity: $\forall x \in \mathcal{AC} : x \geq x$
- Transitivity: $\forall x, y, z \in \mathcal{AC} : x \geq y, y \geq z \Longrightarrow x \geq z$
- Antisymmetry: $\forall x, y \in \mathcal{AC} : x \geq y, y \geq x \Longrightarrow x = y$
- Existence of a least upper bound:

$$\begin{aligned} &\forall x, y \in \mathcal{AC} : \exists! z \in \mathcal{AC} \\ &z \geq x \text{ and } z \geq y \\ &\forall t \in \mathcal{AC} : t \geq x \text{ and } t \geq y \Longrightarrow t \geq z \end{aligned}$$

- Existence of a greatest lower bound:

$$\begin{aligned} &\forall x, y \in \mathcal{AC} : \exists! z \in \mathcal{AC} \\ &x \geq z \text{ and } y \geq z \\ &\forall t \in \mathcal{AC} : x \geq t \text{ and } y \geq t \Longrightarrow z \geq t \end{aligned}$$

Access classes defined as above together with the dominance relationship between them therefore define a lattice [64]. Figure 3.3 determines the security lattice obtained considering security levels *TS* and *S*, with $TS > S$ and the set of categories *{Nuclear, Army}*. The semantics and use of the classifications assigned to objects and subjects within the application of a multilevel mandatory policy is different depending on whether the classification is intended for a *secrecy* or an *integrity* policy.

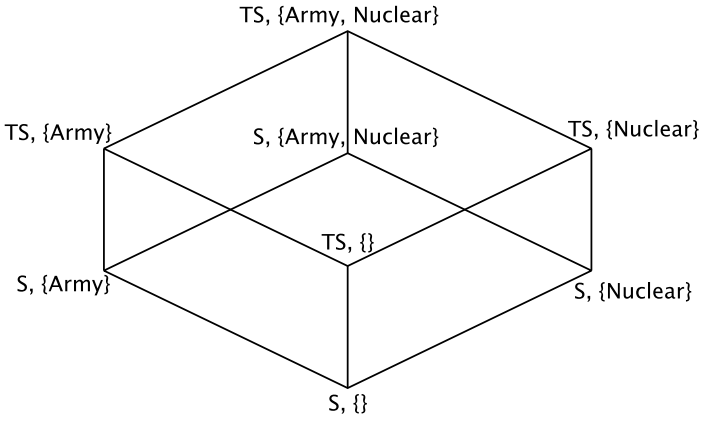

Figure 3.3: Example of security lattice [190]

Secrecy-based Mandatory Policies

A mandatory secrecy policy controls the direct and indirect flows of information to prevent leakages to unauthorized subjects. Here, the semantics of the classification is as follows. The security level of the access class associated with an object reflects the sensitivity of the information contained in the object, that is, the potential damage that could result from the unauthorized disclosure of the information. The security level of the access class associated with a user, also called clearance, reflects the user's trustworthiness not to disclose sensitive information to users not cleared to see it. Categories define the area of competence of users and data and are used to provide finer-grained security classifications of subjects and objects than classifications provided by security levels alone. They are the basis for enforcing restrictions, for instance confining subjects to access the information they need to know to perform their job.

Users can connect to the system at any access class dominated by their clearance. A user connecting to the system at a given access class originates a subject at that access class. For instance, with reference to the lattice in figure 3.3, a user cleared *(TS,{Nuclear})* can connect to the system as a *(S,{Nuclear})*, *(TS,∅)*, or *(TS,∅)* subject. Requests by a subject to access an object are controlled concerning the access class of the subject and the object and granted only if some relationship, depending on the requested access, is satisfied. In particular, two principles, first formulated by Bell and LaPadula [34], have to be satisfied to protect information confidentiality:

- **No-read-up** - A subject S can only read an object O if $S \geq O$ and S has discretionary access to O. Discretionary access means that a subject has clearance to read can be accessed, and make downgrading of a piece of information's security classification impossible. In other words, a subject is allowed a read access to an object only if the access class of the subject dominates the access class of the object.

- **No-write-down** - A subject S can write to an object O if $O \geq S$ and S has discretionary access to O. It means that a subject is allowed a write access to an object only if the access class of the subject is dominated by the access class of the object.

Satisfaction of these two principles prevents information from flowing from high-level subjects/objects to subjects/objects at lower levels, thereby ensuring the satisfaction of the protection requirements, for instance, no process will be able to make sensitive information available to users not cleared for it. It is described in figure 3.4, where four accesses classes composed only of a security level (TS, S, C, and U) are taken as an example. Note the importance of controlling both reads and write operations, since both can be improperly used to leak information.

Regards to the no-write-down principle, it is clear now why users are allowed to connect to the system at different access classes. Thus they can access information at different levels (provided that they are cleared for it). Note that a lower class does not mean „less“ privileges in absolute terms, but only less reading privileges (see figure 3.4).

However users can connect to the system at any level below their clearance, the strict application of the no-read-up and the no-write-down principles may result too rigid. Real world situations often require exceptions to the mandatory restrictions. For instance, data may need to be downgraded, for instance, data subject to embargoes that can be released after some time. Also, information released by a process may be less sensitive than the information the process has read. For instance, a procedure may access personal information

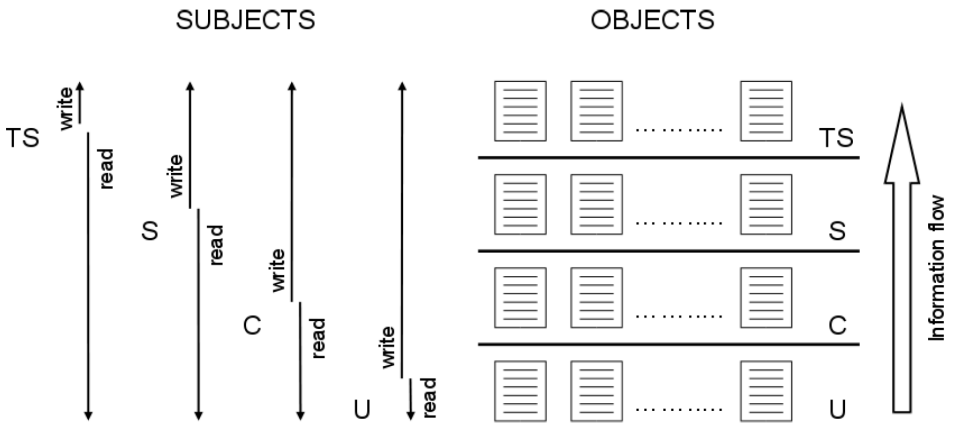

Figure 3.4: Information flow for secrecy [193]

regarding the employees of an organization and return the benefits to be granted to each employee. While the personal information can be considered secret, the benefits can be considered Confidential. In order to respond to situations like these, multilevel systems should then allow for exceptions, loosening or waiving restrictions, in a controlled way, to processes that are trusted and ensure that information is sanitized (meaning the sensitivity of the original information is lost).

Note also that DAC and MAC policies are not mutually exclusive, but can be applied jointly. In this case, access to be granted needs both, the existence of the necessary authorization for it, and also to satisfy the mandatory policy. Intuitively, the discretionary policy operates within the boundaries of the mandatory policy - it can only restrict the set of accesses that would be allowed by mandatory access control alone.

The secrecy based control principles just illustrated summarize the basic axioms of the security model proposed by David Bell and Leonard LaPadula[34]. There are some concepts of the model formalization to give an idea of the different aspects to be taken into account in the definition of a security model. In this model a system is composed of a set of subjects S, objects O, and actions A, which includes *read* and *write*. A *write* actions in this context behave as write-only or append operation.

The model also assumes a lattice L of access classes and a function: $\lambda : S \cup O \rightarrow L$ that, when applied to a subject (object respectively) in a given state, returns the classification of the subject (object respectively) in that state. A state $v \in V$ is defined as a triple (b, M, λ), where $b \in \mathcal{P}(S \times O \times A)$ is the set of current accesses (s, o, a), M is the access matrix expressing discretionary permissions same as in the HRU model, and λ is the association of access classes with subjects and objects. A system consists of an initial state v_0, a set of requests R, and a state transition function $T : V \times R \rightarrow V$ that transforms a system state into another state resulting from the execution of the request. Intuitively, requests capture acquisition and release of accesses, granting and revocation of authorizations, as well as changes of levels.

The model then defines a set of axioms stating properties that the system must satisfy and that express the constraints imposed by the mandatory policy. The first version of the Bell and LaPadula model stated the following criteria:

- **simple property** - A state v satisfies the simple security property if $\forall s \in S, o \in O : (s, o, \text{read}) \in b \Rightarrow \lambda(s) \geq \lambda(o)$.
- ***-property** - A state v satisfies the **-security* property if $\forall s \in S, o \in O : (s, o, \text{write}) \in b \Rightarrow \lambda(o) \geq \lambda(s)$.

The two axioms above correspond to the no-read-up and no-write-down principles that were described in section 3.4. A state is then defined to be secure if it satisfies both the simple security property and the **-property*. A system (v_0, R, T) is secure if and only if every state reachable from v_0 by executing one or more finite sequences of requests from R is *state secure*. In the first formulation of their model, Bell and LaPadula provide a *basic security theorem* (BST), which states that a system is secure when its initial state v_0 is secure, and also the state transition T is security preserving, that is, it transforms a secure state into another secure state.

Integrity-based Mandatory Policies

The mandatory policy that was discussed above protects only the confidentiality of the information, and there is no control enforced on its integrity. Low classified subjects could still be able to enforce improper indirect modifications to objects they cannot write. Starting from the principles of the Bell and LaPadula model, Biba [40] proposed a dual policy for safeguarding the integrity, which controls the flow of information and prevents subjects from modifying information they cannot write indirectly.

Similarly, as for secrecy, each subject and object in the system is assigned an integrity classification. The classifications and the dominance relationship between them are defined as before. Example of integrity levels can be: high important (HI), medium important (MI), and low important (LI). The semantics of integrity classifications is as follows. The integrity level associated with a user reflects the user's trustworthiness for inserting, modifying, or deleting information. The integrity level associated with an object reflects both the degree of trust that can be placed on the information stored in the object and the potential damage that could result from unauthorized modifications of the information. Again, categories define the area of competence of users and data. Access control is enforced according to the following two principles:

- **No-read-down** - A subject is allowed to a read access to an object only if the access class of the object dominates the access class of the subject.
- **No-write-up** - A subject is allowed to a write access to an object only if the access class of the subject dominates the access class of the object.

Satisfaction of these principles safeguards integrity by preventing information stored in low objects (and therefore less reliable) to flow to higher, or incomparable, objects. This principle is illustrated in figure 3.5, where classes composed only of integrity levels (HI, MI, and LI) are taken as an example.

Role-based Access Control

Role-based access control (RBAC) is an alternative to traditional DAC and MAC policies that are attracting increasing attention, particularly for commercial applications. The main motivation behind RBAC is the necessity to specify and enforce enterprise-specific security

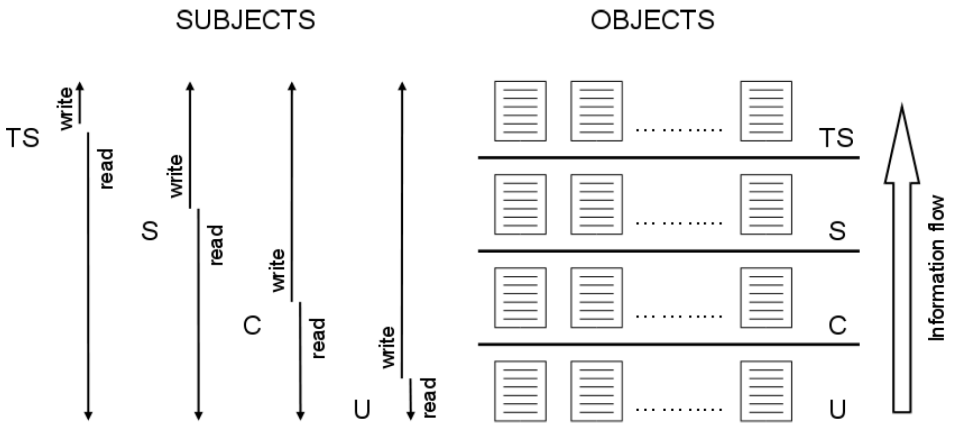

Figure 3.5: Information flow for integrity [193]

policies in a way that maps naturally to an organization's structure. In fact, in a large number of business activities, a user's identity is relevant only from accountability. For access control purposes it is much more important to know what a user's organizational responsibilities are, rather than who the user is. The conventional discretionary access controls, in which individual user ownership of data plays such an important part, are not a good fit. Neither is the full mandatory access controls, in which users have security clearances, and objects have security classifications. Role-based access control tries to fill in this gap by merging the flexibility of explicit authorizations with additionally imposed organizational constraints.

Essentially, role-based policies require the identification of *roles* in the system, where a role can be defined as a set of actions and responsibilities associated with a particular working activity. The role can be widely scoped, reflecting a user's job title, for instance, *secretary*, or it can be more specific, reflecting, for instance, a task that the user needs to perform, such as *order processing*. Then, instead of specifying all the accesses, each user is allowed to execute, access authorizations on objects are specified for roles. Users are then given authorizations to adopt roles (see figure 3.6). The user playing a role is allowed to execute all accesses for which the role is authorized. In general, a user can take on different roles on different occasions. Also, the same role can be played by several users, perhaps simultaneously.

It is important to note the difference between groups and roles. Groups define sets of users while roles define sets of privileges. There is a semantic difference between them - roles can be „activated“ and „deactivated“ by users at their discretion, while group membership always applies, that is, users cannot enable and disable group memberships and corresponding authorizations at their will. However, since roles can be defined which correspond to organizational figures, for instance, *secretary, chair*, and *faculty*, the same mechanism can be seen both as a group and as a role. The role-based approach has several advantages. Some of these are discussed below.

- **Authorization management** - Role-based policies benefit from a logical independence in specifying user authorizations by breaking this task into two parts - assignment of roles to users, and assignment of authorizations to access objects to roles. It greatly simplifies the management of the security policy. When a new user joins

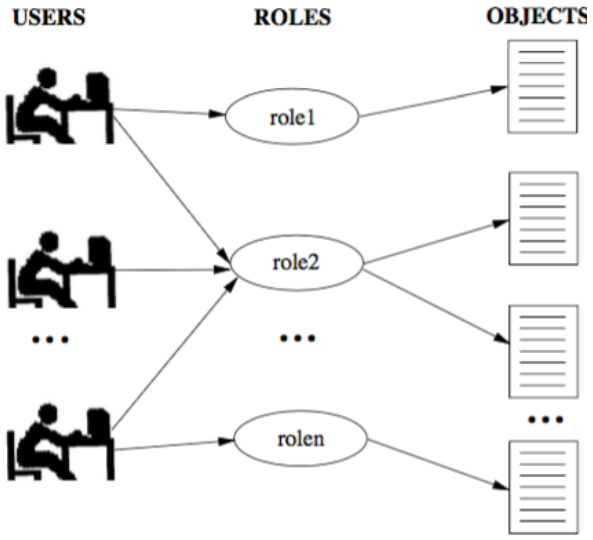

Figure 3.6: Role-based access control [190]

the organization, the administrator only needs to grant her the roles corresponding to her job. If afterward a user's job changes, the administrator simply has to change the roles associated with that user. When a new application or task is added to the system, the administrator needs only to decide which roles are permitted to execute it.

- **Hierarchical roles** - In many applications, there is a natural hierarchy of roles, based on the familiar principles of generalization and specialization. Figure 3.7 illustrates an example of role hierarchy. Each role is represented as a node, and there is an arc between a specialized role and its generalization. The role hierarchy can be exploited for authorization implication. For instance, authorizations granted to roles can be propagated to their specializations, such as the *secretary* role can be allowed all accesses granted to *adm-staff*. Authorization implication can also be enforced on role assignments, by allowing users to activate all generalizations of the roles assigned to them. For instance, a user allowed to activate *secretary* will also be allowed to activate role *adm-staff*. Authorization implication has the advantage of further simplifying authorization management. Note however that not always implication may be wanted, as propagating all authorizations is contrary to the least privilege principle.

- **Least privilege** - Roles allow a user to sign on with the least privilege required for the particular task she needs to perform. Users authorized to powerful roles do not need to exercise them until those privileges are needed. It minimizes the possible danger of damage due to inadvertent errors, or intruders masquerading as legitimate users.

- **Separation of duties** - Separation of duties refer to the principle that no user should be given enough privileges to misuse the system on their own. For instance, the person authorizing a paycheck should not be the same person who can prepare them. Separation of duties can be enforced either statically by defining conflicting roles, that is, roles which cannot be executed by the same user or dynamically by

enforcing the control at access time. An example of dynamic separation of duty restriction is the two-person rule. The first user to execute a two-person operation can be an authorized user, whereas the second user can be any authorized user different from the first.

- **Constraints enforcement** - Roles provide a basis for the specification and enforcement of further protection requirements that real-world policies may need to express. For instance, cardinality constraints can be specified, that restrict the number of users allowed to activate a role or the number of roles allowed to exercise a given privilege. The constraints can also be dynamic, that is, be imposed on roles activation rather than on their assignment. For instance, while several users may be allowed to activate role *chair*, a further constraint can require that at most one user at a time can activate it.

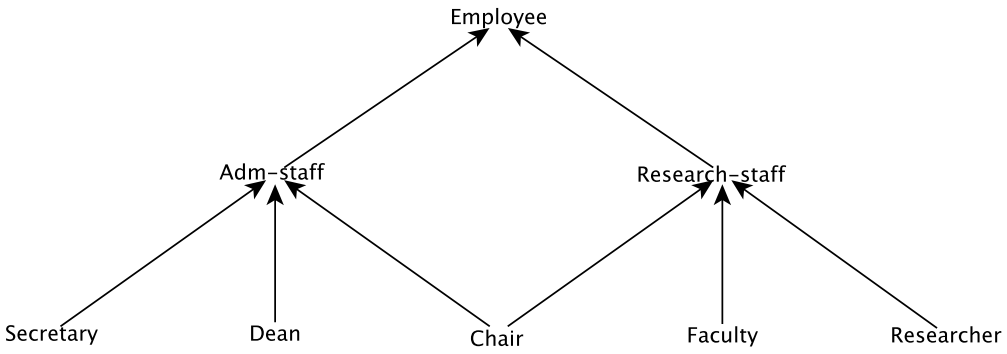

Figure 3.7: An example of hierarchy [190]

Role-based Access Control Model

The role-based access control model [78] is defined regarding four *model components*: core RBAC, hierarchical RBAC, static separation of duty relations and dynamic separation of duty relations. Core RBAC defines a minimum collection of elements, element sets, and relations to completely achieve a role-based access control system. It includes user-role assignment and permission-role assignment relations, considered fundamental in an RBAC system. Moreover, core RBAC introduces the concept of role activation as part of user's session within a computer system. The core of role-based access control is required in any RBAC system, but the other components are independent of each other and may be implemented separately or not even used. Each model component is defined by these sub-components:

- A set of basic element sets.
- A set of RBAC relations involving those element sets (containing subsets of Cartesian products denoting valid assignments).
- A set of mapping functions that yield instances of members from one element set for a given instance from another element set.

Core role-based access control model element sets and relations are defined in figure 3.8. Core includes sets of five basic data elements called users (*USERS*), roles (*ROLES*), objects (*OBJ*), operations (*OPS*), and permissions (*PRMS*). The model as a whole is fundamentally defined regarding individual users being assigned to roles and permissions being assigned to roles. As such, a role is a means for naming many-to-many relationships among individual users and permissions. Moreover, the core model includes a set of sessions (*SESSIONS*) where each session is a mapping between a user and an activated subset of roles that are assigned to the user.

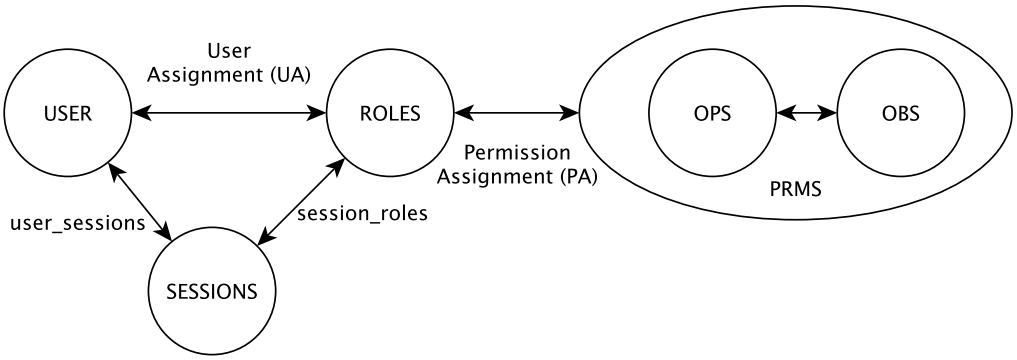

Figure 3.8: The core of role-based access control [78]

A *user* is defined as a human being. Although the concept of a user can be extended to include machines, networks, or intelligent autonomous agents, for simplicity reasons the description is limited to a user as a person. A *role* is a job function within the context of an organization with some associated semantics regarding the authority and responsibility conferred on the user assigned to the role. *Permission* is an approval to operate on one or more protected objects. An *operation* is an executable image of an application, which upon invocation executes some function for the user. The types of operations and objects that RBAC controls are dependent on the type of system in which they will be implemented. For instance, within a file system, operations might include read, write, and execute, on the other type of system such as a database management system, operations might include insert, delete, append, and update.

The purpose of any access control mechanism is to protect system resources. However, in applying RBAC to a computer system, the description is about protecting objects. Consistent with earlier models of access control an object is an *entity* that contains or receives information. For a system that implements RBAC, the objects can represent information containers, for instance, files or directories in an operating system, and columns, rows, tables, and views within a database management system. An *objects* can represent exhaustible system resources, such as printers, disk space, and CPU cycles. The set of *objects* covered by RBAC includes all of the *objects* listed in the permissions that are assigned to roles.

Central to RBAC is the concept of role relations, around which a role is a semantic construct for formulating policy. Figure 3.8 illustrates user assignment (*UA*) and permission assignment (*PA*) relations. The arrows indicate a many-to-many relationship, for instance, a user can be assigned to one or more roles, and a role can be assigned to one or more users. This arrangement provides great flexibility and granularity of assignment of permissions

to roles and users to roles. Without these conveniences, there is an enhanced danger that a user may be granted more access to resources that is needed because of limited control over the type of access that can be associated with users and resources. Users may need to list directories and modify existing files, for example, without creating new files, or they may need to append records to a file without modifying existing records. Any increase in the flexibility of controlling access to resources also strengthens the application of the principle of least privilege.

Each session is a mapping of one user to possibly many roles, that is, a user establishes a session during which the user activates some subset of roles that he or she is assigned. Each session is associated with a single user, and each user is associated with one or more sessions. The function *session_roles* gives information about the roles activated by the session, and the function user sessions give us the set of sessions that are associated with a user. The permissions available to the user are the permissions assigned to the roles that are activated across all the user's sessions. Figure 3.8 can be described in the following definitions 3.4.9.

Definition 3.4.9. Core role-based access control [78]

- $USERS, ROLES, OPS$, and OBS the required sets, user, roles, operations, and objects, respectively.
- $UA \subseteq USERS \times ROLES$, a many-to-many mapping user-to-role assignment relation.
- $assigned\_users : (r : ROLES) \rightarrow 2^{USERS}$, the mapping of role r onto a set of users. Formally: $assigned\_users(r) = \{u \in USERS | (u, r) \in UA\}$.
- $PRMS = 2^{(OPS \times OBS)}$, the set of permissions.
- $PA \subseteq PRMS \times ROLES$, a many-to-many mapping permission-to-role assignment relation.
- $assigned\_permissions(r : ROLES) \rightarrow 2^{PRMS}$, the mapping of role r onto a set of permissions. Formally: $assigned\_permissions(r) = \{p \in PRMS | (p, r) \in PA\}$.
- $Op(p : PRMS) \rightarrow \{op \subseteq OPS\}$, the permission-to-operation mapping, which gives the set of operations associated with permission p.
- $Ob(p : PRMS) \rightarrow \{ob \in OBS\}$, the permission-to-object mapping, which gives the set of objects associated with permission p.
- $SESSIONS$, the set of sessions.
- $user\_sessions(u : USERS) \rightarrow 2^{SESSIONS}$, the mapping of user u onto a set of sessions.
- $session\_roles(s : SESSIONS) \rightarrow 2^{ROLES}$, the mapping of sessions s onto a set of roles. Formally: $session\_roles(s_i) \subseteq r \in ROLES | (session\_users(s_i), r) \in UA)$.
- $available\_session\_perms(s : SESSIONS) \rightarrow 2^{PRMS}$, the permissions available to a user in a session, $\bigcup_{r \in session\_roles(s)} assigned\_permissions(r)$.

As was already described, the basic element sets in core RBAC are *USERS, ROLES, OPS*, and *OBS*. Of these element sets, *OPS* and *OBS* are considered predefined by the underlying system for which is the access control deployed. For instance, a banking system may have predefined transactions (*OPS*) for savings deposit and others, and predefined data sets (*OBS*) such as saving files, addresses, and other necessary data. Administrators create and delete *USERS* and *ROLES*, and establish relationships between roles and existing operations and objects. Required function definitions (taken from RBAC standard [78]), which creates the complete functional specification of this access control mechanism follows. The notation used in the formal specification of the RBAC requirements is a subset of Z notation [206]. The only major change is the representation of a schema:
$Schema - Name(Declaration)\{Predicate; \ldots ; Predicate\}$.

Definition 3.4.10. Add user [78]

$AddUser(user : NAME)\{$
$\quad user \notin USERS$
$\quad USERS' = USERS \cup user$
$\quad user\_sessions' = user\_sessions \cup user \rightarrow \emptyset\}$

Definition 3.4.11. Delete user [78]

$DeleteUser(user : NAME)\{$
$\quad user \in USERS$
$\quad [\forall s \in SESSIONS \bullet s \in user\_sessions(user) \Rightarrow DeleteSession(s)]$
$\quad UA' = UA \setminus \{r : ROLES \bullet user \rightarrow r\}$
$\quad assigned\_users' = \{r : ROLES \bullet r \rightarrow (assigned\_users(r) \setminus \{user\})\}$
$\quad USERS' = USERS \setminus \{user\}\}$

Definition 3.4.12. Add role [78]

$AddRole(role : NAME)\{$
$\quad role \notin ROLES$
$\quad ROLES' = ROLES \cup \{role\}$
$\quad assigned\_users' = assigned\_users \cup \{role \rightarrow \emptyset\}$
$\quad assigned\_permissions' = assigned\_permissions \cup \{role \rightarrow \emptyset\}\}$

Definition 3.4.13. Delete role [78]

$$
\begin{aligned}
&DeleteRole(role : NAME)\{\\
&\quad role \in ROLES\\
&\quad [\forall s \in SESSIONS \bullet role \in session\_roles(s) \Rightarrow DeleteSessions(s)]\\
&\quad UA' = UA \setminus \{u : USERS \bullet u \rightarrow role\}\\
&\quad assigned\_users' = assigned\_users \setminus \{role \rightarrow assigned\_users(role)\}\\
&\quad PA' = PA \setminus \{op : OPS, ob : OBS \bullet (op, ob) \rightarrow role\}\\
&\quad assigned\_permissions' = assigned\_permissions \setminus\\
&\qquad \{role \rightarrow assigned\_permissions(role)\}\\
&\quad ROLES' = ROLES \setminus \{role\}\}
\end{aligned}
$$

Definition 3.4.14. Assign user [78]

$$
\begin{aligned}
&AssignUser(user, role : NAME)\{\\
&\quad user \in USERS; role \in ROLES; (user \rightarrow role) \notin UA\\
&\quad UA' = UA \cup \{user \rightarrow role\}\\
&\quad assigned\_users' = assigned\_users \setminus \{role \rightarrow assigned\_users(role)\} \cup\\
&\qquad\qquad \{role \rightarrow (assigned\_users(role) \cup \{user\})\}\}
\end{aligned}
$$

Definition 3.4.15. Deassign user [78]

$$
\begin{aligned}
&DeassignUser(user, role : NAME)\{\\
&\quad user \in USERS; role \in ROLES; (user \rightarrow role) \in UA\\
&\quad [\forall s : SESSIONS \bullet s \in user\_sessions(user) \wedge role \in session\_roles(s)\\
&\qquad \Rightarrow DeleteSessions(s)]\\
&\quad UA' = UA \setminus \{user \rightarrow role\}\\
&\quad assigned\_users' = assigned\_users \setminus \{role \rightarrow assigned\_users(role)\} \cup\\
&\quad \{role \rightarrow (assigned\_users(role) \setminus \{user\})\}\}
\end{aligned}
$$

Other relevant definitions, such as grant permission, revoke permission, create a session, delete a session, add an active role, drop an active role, and many others are described in the RBAC standard by D. F. Ferraiolo et al. [78].

3.5 Virtual Machine

The term virtualization has many meanings, and aspects of virtualization permeate all aspects of computing. Virtual machines are one instance of this trend. Generally, with a virtual machine, guest operating systems and applications run in an environment that appears to them to be native hardware, and that behaves toward them as native hardware would, but that also protects, manages, and limits them. This section delves into the uses, features, and implementation of virtual machines. Virtual machines can be implemented in several ways, and this section describes these options. One option is to add virtual machine support to the kernel. Additionally, hardware features provided by the CPU and

even by I/O devices can support virtual machine implementation, so it is discussed how those features are used by the appropriate kernel modules.

The fundamental idea behind a virtual machine is to abstract the hardware of a single computer (the CPU, memory, disk drives, and network interface cards) into several different execution environments, thereby creating the illusion that each separate environment is running on its private computer. In the case of virtualization, there is a layer that creates a virtual system on which operating systems or applications can run. Virtual machine implementations involve several components. At the base is the host, the underlying hardware system that runs the virtual machines. The virtual machine manager (VMM) (also known as a *hypervisor*) creates and runs virtual machines by providing an interface that is identical to the host. Each guest process is provided with a virtual copy of the host (see figure 3.9). Usually, the guest process is, in fact, an operating system. A single physical machine can thus run multiple operating systems concurrently, each in its virtual machine.

Note that with virtualization, the definition of „operating system“ blurs. For instance, consider VMM software such as VMware ESX [224]. This virtualization software is installed on the hardware, run when the hardware boots, and provides services to applications. The services include traditional ones, such as scheduling and memory management, along with new types, such as migration of applications between systems. Furthermore, the applications are in fact guest operating systems. Is the VMware ESX VMM an operating system that, in turn, runs other operating systems? Certainly, it acts as an operating system. For clarity, however, it is called the component that provides virtual environments a VMM. The implementation of VMMs varies greatly. Options include the following:

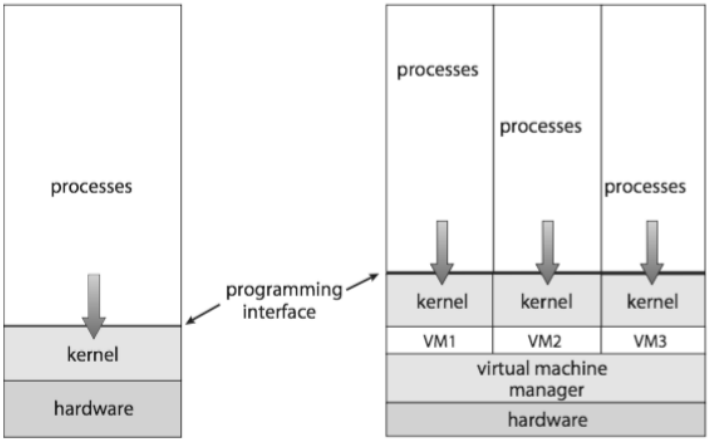

Figure 3.9: Non-virtual machine (left), Virtual machine (right) [175]

- Hardware-based solutions that provide support for virtual machine creation and management through firmware. These VMMs, which are commonly found in mainframe and large to mid-sized servers, are generally known as *type 0 hypervisors*. The examples of these systems are IBM LPARs [114] and Oracle LDOMs [155].
- Operating-system-like software built to provide virtualization, including VMware ESX, Joyent SmartOS [99], and Citrix XenServer [220]. These VMMs are known as *type 1 hypervisors*.

- General-purpose operating systems that provide standard functions as well as VMM functions, including Microsoft Windows Server [187] with HyperV [219] and RedHat Linux [177] with the KVM [124] feature. Because such systems have a feature set similar to type *1 hypervisors*, they are also known as *type 1*.
- Applications that run on standard operating systems, but provide VMM features to guest operating systems. These applications, which include VMware Workstation [48] and Fusion, Parallels Desktop [162], and Oracle Virtual-Box [170], are *type 2 hypervisors*.

The variety of virtualization techniques in use today is a testament to the breadth, depth, and importance of virtualization in modern computing. Virtualization is invaluable for data-center operations, efficient application development, and software testing, among many other uses.

Features of Virtual Machine

Several advantages make virtualization attractive. Most of them are fundamentally related to the ability to share the same hardware yet run several different execution environments (that is, different operating systems) concurrently. One crucial advantage of virtualization is that the host system is protected from the virtual machines, just as the virtual machines are protected from each other. A virus inside a guest operating system might damage that operating system but is unlikely to affect the host or the other guests. Because each virtual machine is almost completely isolated from all other virtual machines, there are almost no protection problems. A potential disadvantage of isolation is that it can prevent sharing of resources. Two approaches to providing sharing have been implemented. First, it is possible to share a file-system volume and thus to share files. Second, it is possible to define a network of virtual machines, each of which can send information over the virtual communications network. The network is modeled after physical communication networks but is implemented in software. Of course, the VMM is free to allow any number of its guests to use physical resources, such as a physical network connection (with sharing provided by the VMM), in which case the allowed guests could communicate with each other via the physical network.

One feature common to most virtualization implementations is the ability to *freeze*, or *suspend*, a running virtual machine. Many operating systems provide that basic feature for processes, but VMMs go one step further and allow copies and snapshots to be made of the guest. The copy can be used to create a new VM or to move a VM from one machine to another with its current state intact. The guest can then resume where it was, as if on its original machine, creating a clone. The snapshot records a point in time, and the guest can be reset to that point if necessary (for example, if a change was made but is no longer wanted). Often, VMMs allow many snapshots to be taken. For instance, snapshots might record a guest's state every day for a month, making restoration to any of those snapshot states possible. These abilities are used to good advantage in virtual environments.

A virtual machine system is a perfect vehicle for operating-system research and development. Typically, changing an operating system is a difficult task. Operating systems are large and complex programs, and a change in one part may cause obscure bugs to appear in some other part. The power of the operating system makes changing it particularly dangerous. Because the operating system executes in kernel mode, a wrong change in a pointer

could cause an error that would destroy the entire file system. Thus, it is necessary to test all changes to the operating system carefully.

Furthermore, the operating system runs on and controls the entire machine, meaning that the system must be stopped and taken out of use while changes are made and tested. This period is commonly called system-development time. Since it makes the system unavailable to users, system-development time on shared systems is often scheduled late at night or on weekends, when system load is low.

A virtual-machine system can eliminate much of this latter problem. System programmers are given their virtual machine, and system development is done on the virtual machine instead of on a physical machine. Normal system operation is disrupted only when a completed and tested change is ready to be put into production.

Another advantage of virtual machines for developers is that multiple operating systems can run concurrently on the developer's workstation. This virtualized workstation allows for rapid porting and testing of programs in varying environments. Besides, multiple versions of a program can run, each in its isolated operating system, within one system. Similarly, quality- assurance engineers can test their applications in multiple environments without buying, be powering, and maintain a computer for each environment.

A significant advantage of virtual machines in production data-center use is system consolidation, which involves taking two or more separate systems and running them in virtual machines on one system. Such physical-to-virtual conversions result in resource optimization since many lightly used systems can be combined to create one more heavily used a system.

Consider, too, that management tools that are part of the VMM allow system administrators to manage many more systems than they otherwise could. A virtual environment might include 100 physical servers, each running 20 virtual servers. Without virtualization, 2,000 servers would require several system administrators. With virtualization and its tools, the same work can be managed by one or two administrators. One of the tools that make this possible is templating, in which one standard virtual machine image, including an installed and configured guest operating system and applications, is saved and used as a source for multiple running VMs. Other features include managing the patching of all guests, backing up and restoring the guests, and monitoring their resource use.

Virtualization can improve not only resource utilization but also resource management. Some VMMs include a live migration feature that moves a running guest from one physical server to another without interrupting its operation or active network connections. If a server is overloaded, live migration can thus free resources on the source host while not disrupting the guest. Similarly, when host hardware must be repaired or upgraded, guests can be migrated to other servers, the evacuated host can be maintained, and then the guests can be migrated back. This operation occurs without downtime and interruption to users.

Think about the possible effects of virtualization on how applications are deployed. If a system can quickly add, remove, and move a virtual machine, then why install applications on that system directly? Instead, the application could be pre-installed on a tuned and customized operating system in a virtual machine. This method would offer several benefits for application developers. Application management would become more comfortable, less tuning would be required, and technical support of the application would be more straightforward. System administrators would find the environment easier to manage as well. Installation would be simple, and redeploying the application to another system would be much more comfortable than the usual steps of uninstalling and re-installing. For widespread adoption of this methodology to occur, though, the format of virtual machines

must be standardized so that any virtual machine will run on any virtualization platform. The „Open Virtual Machine Format; is an attempt to provide such standardization, and it could succeed in unifying virtual machine formats.

Virtualization has laid the foundation for many other advances in computer facility implementation, management, and monitoring. Cloud computing, for example, is made possible by virtualization in which resources such as CPU, memory, and I/O are provided as services to customers using Internet technologies. By using APIs, a program can tell a cloud computing facility to create thousands of VMs, all running a specific guest operating system and application, which others can access via the Internet. Many multiuser games, photo-sharing sites, and other web services use this functionality.

In the area of desktop computing, virtualization is enabling desktop and laptop computer users to connect remotely to virtual machines located in remote data centers and access their applications as if they were local. This practice can increase security because no data are stored on local disks at the user's site. The cost of the user's computing resource may also decrease. The user must have networking, CPU, and some memory, but all that these system components need to do is display an image of the guest as its runs remotely. Thus, they need not be expensive, high-performance components. Other uses of virtualization are sure to follow as it becomes more prevalent and hardware support continues to improve.

3.6 Summary

This chapter discussed the protection on operating system level and the theory about the protection mechanisms such as access control principles, models, and virtualization. The most of these protection mechanisms were presented formally. These mechanisms are the cornerstone of the security through all platforms. The decision which of these principles are used is the provider of the platform. These principles are heavily used and modified for the specific purposes on mobile devices.

Operating systems designed for mobile devices required a different approach compared to operating systems designed for desktop computers. Moreover, implementation of selected mechanism can differ from one vendor of an operating system to another. Implementation solution should be confirmed by the verification process with the model of required behavior or in this case the model definition of for example access control. The main difference is the ability to save as much power as possible, which has an impact on almost every aspect of the operating system. Algorithms are redesigned to consume less memory as possible, and the user experience persists.

The last part of this chapter discusses the virtual machine and its features. The virtual machine is a critical part of the operating system of a mobile world where it is used. The next chapter discusses the mechanism presented in this chapter applied in the world of mobile operating systems on a selected mobile platform.

Chapter 4

Mobile Platform Architecture

This chapter consists of the description of open source available mobile platform architecture. It is focused on open source solution because implementation details are part of already presented principles. According to the aim of the thesis, the Android platform has been chosen as the reference platform. The main reasons for this choice are open source, large community, possibility to modify the system and test it in the simulation environment or the real devices, and other reasons related to law and trademarks. Everything that the platform offers is free of charge.

The mobile platform architecture follows up the previous chapter, which was about protection of operating systems. There is more detailed information about the mentioned concepts, and the details are targeted to the Android platform, which is currently the most popular and widespread operating system on the world [229].

Android is an application execution platform for mobile devices comprised out of an operating system, core libraries, development framework and necessary applications. The Android architecture stack contains the whole platform levels which correspond to security levels. The overall architecture is illustred in figure 4.1.

Android operating system is built on top of a Linux kernel. The Linux kernel is responsible for executing core system services such as memory access, process management, access to the physical device through drivers, network management, and security. Atop the Linux kernel is the virtual machine called Dalvik virtual machine [168, 70] (or the successor of Dalvik called Android runtime (ART) virtual machine [56]) along with necessary system libraries. The Dalvik/ART virtual machine is a register-based execution engine used to run Android applications.

Android is comprised of several mechanisms playing a role in the security checking and enforcement. Like any modern operating system, many of these tools interact with each other, exchanging information about subjects (applications/users), objects (different applications, files, devices), and operations to be performed (read, write, and delete). Frequently, enforcement occurs without incident, but occasionally, things slip through the cracks, affording an opportunity for abuse. This chapter discusses the security design and architecture of chosen Android platform.

4.1 Virtual Machine

To achieve run-time support a diverse set of mobile devices and applications have to be sandboxed for security, performance, and reliability, a virtual machine is a distinct tech-

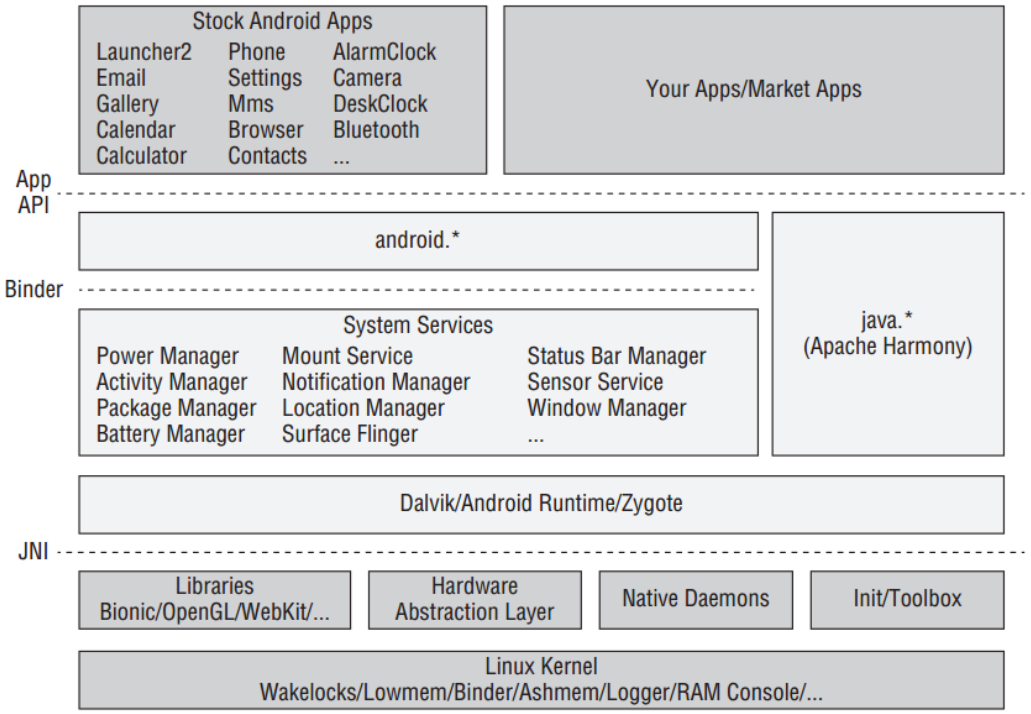

Figure 4.1: General Android system architecture [68]

nology to be used. The virtual machine does not necessarily satisfy the requirements with the limited processor power and also limited memory, that characterize most mobile devices. Virtual machine developers have favored stack-based architecture over register-based architectures. It is mostly due to the simplicity of implementation, ease of writing a compiler back-end. Virtual machines are originally designed to host a single language and density. Executable applications are invariably smaller than applications for register-based architectures. The simplicity and code density comes at the cost of performance.

Given that the virtual machine is running on devices with constrained processing power, the choice of a register-based architecture seems appropriate. The virtual machine on Android platform relies on the Linux kernel for underlying functionality such as threading and low-level memory management. Each application runs in its process with its instance of the virtual machine. Implementation has been written so that a device can run multiple instances of virtual machines efficiently. The overall architecture of the application package which has the compressed behavior in the form of *dex* file, which is similar to a collection of compressed *class* data (compiled java source code). This *dex* file is the input for the virtual machine.

ART is an application run-time environment provides the execution of applications on the Android operating system. It is the follower of the previous version of the virtual machine for this platform - Dalvik. Replacing Dalvik performs the translation of the application's byte-code into native instructions that are later executed by the run-time environment.

The history of evolution starts with the trace-based just-in-time compilation in Dalvik, optimizing the execution of applications by continually profiling forms each time they are processed and dynamically compiling frequently executed short segments of their byte-

code into native machine code. While Dalvik interprets the rest of the application's byte-code, native execution of those short byte-code parts, called traces, provides significant performance improvements.

In contrast, the ART introduces the use of ahead-of-time compilation by compiling entire applications into native machine code upon their installation. By eliminating interpretation of trace-based just-in-time compilation, it improves the execution efficiency and reduces the power consumption, which results in enhanced battery on mobile devices. At the same time, ART brought faster execution of applications, improved memory allocation and garbage collection mechanisms, and more accurate profiling of applications. To keep backward compatibility, ART uses the same input byte-code format as Dalvik, supplied through *dex* files as part of the installation package.

At the last version (currently 8.0) of the Android operating system, the just-in-time compiler introduced with improvements related to code profiling into ART, which continuously improve the performance of Android applications as they run.

4.2 Sandbox

The model based on application isolation in a sandbox environment. It means that each application executes in its environment and is unable to influence or modify execution of any other application. Application sandboxing is performed at the Linux kernel level. To achieve isolation, Android utilizes standard Linux access control mechanisms. Each application installation package *apk* is during installation assigned with a unique user identification number (user ID). This approach allows the platform to enforce standard file access rights as it is known from Linux based operating systems. Since each file is associated with its owner's user ID, applications cannot access files that belong to other applications without being granted appropriate permissions. Each file can be assigned read, write and execute access permission. Since the root/administrator user owns the system files, applications are not able to act maliciously by accessing or modifying critical system components. On the other hand, to achieve memory isolation, every application is running in its process (see figure 4.2), i.e., each application has its memory space assigned. Additional security is achieved by utilizing memory management unit (MMU) [88], a hardware component used to translate between virtual and physical address space. This way an application can not compromise system security by running native code in privileged mode, i.e., the application is unable to modify the memory segment assigned to the operating system.

The presented isolation model provides a secure environment for application execution. However, restrictions enforced by the model also reduce the overall application functionality. For example, useful features could be achieved by accessing critical systems such as access to network services, camera or location services. Furthermore, exchange of a data and functionalities between applications enhanced the capabilities of the development framework. The shared user ID and permissions are two mechanisms, introduces by the Android platform, which can be used to lift the restrictions enforced by the isolation model.

The mechanism must provide sufficient flexibility to the application developers, but also preserve the overall system security. Two applications can share data and application components, i.e., activities, content providers, services and broadcast receivers. For example, an application can run an activity belonging to other application or access its files. The shared user ID allows applications to share data and application components. To be assigned with a shared user ID the two applications must be signed with the same digital certificate. In effect, the developers can bypass the isolation model restrictions by signing

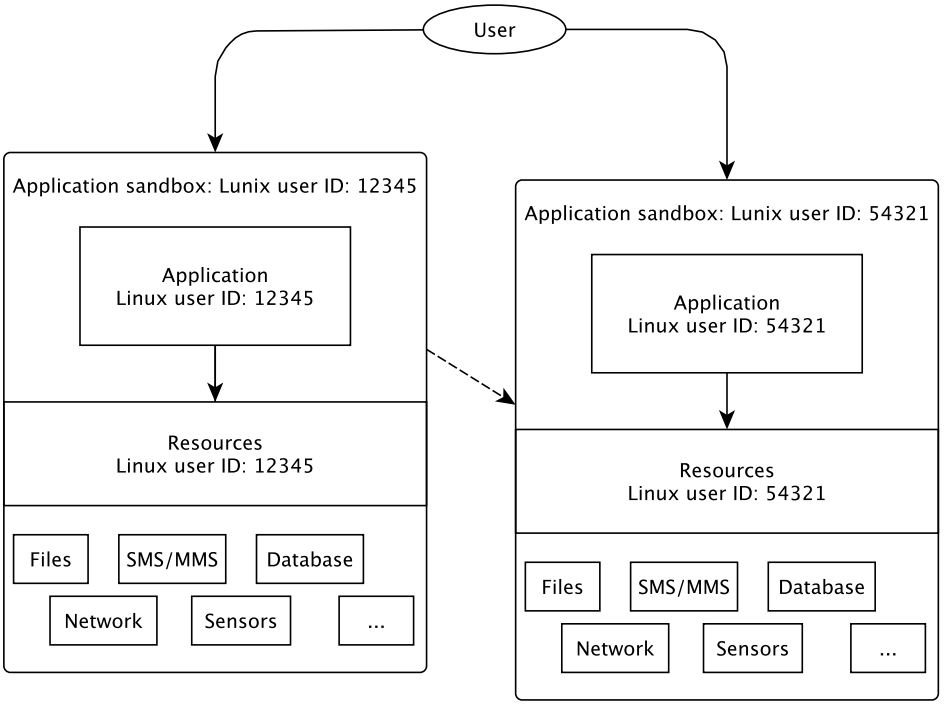

Figure 4.2: Two Android applications, each on its own sandbox

applications with the corresponding private key. This approach is not recommended to use usually, but in specific cases only. However, since there is not a central certification authority, the developers are responsible for keeping their private keys secure. By sharing the user ID, applications gain the ability to run in the same process. The recommended alternative to the shared user ID approach is to use Android permissions. In addition to sharing data and components, the permissions are used to gain access to critical system modules. Each application can request and define a set of permissions. It means that each application can expose a subset of its functionality to other applications if they have been granted the corresponding permissions. Besides, each application can request a set of permissions to access other applications or system libraries.

Permissions are granted by the operating system during installation and can be changed afterward manually. There are four types of permissions: normal, dangerous, signature and signature-or-system. Standard permission give access to isolated application-level functionalities. These functionalities have little impact on the system or user security and are therefore granted without an explicit user's approval. The following section describes permissions in more detail.

4.3 Permissions

However, the user can review which permissions are requested before application installation, he must agree with all requested permission, or the installation is aborted. As was discussed in the previous section there are four groups or types of permissions. An example of a *normal* level permission is access to the phone's vibration hardware unit. Since it is an isolated functionality, i.e., user's privacy or other applications cannot be compromised,

it is not considered an impact on the system in the area of security. On the other hand, a *dangerous* level permission provides access to private data and critical system functionality. For example, by obtaining a dangerous permission, an application can use telephony services, network access, location information or gain access to other private data. Since a *dangerous* permission level presents a high-security risk, the user is prompted to confirm set of requested permissions before installation of an application. Android has the all-or-nothing architecture for permission granting in the meaning of installation of applications. There is a possibility to change permissions in the settings after the successful installation process.

Applications can access only the resources for which they have permission. Further, it is observed that most of the applications ask for more permissions then they needed. They can misuse it for malicious activities and information leakage. Signature permission level can be granted to the application signed with the same certificate as application declaring the permission. The signature permission level is in effect a refinement of the shared user ID approach and provides more control in sharing application data and components. On the other hand, signature-or-system permission level extends the signature permission level by granting access to the applications installed in the Android system image [93]. However, caution is required since both the signature and signature-or-system permissions will allow access rights without asking for the user's explicit approval.

Permission Model

This section formally specifies the Android permission scheme by identifying the system elements and describing their relationships. The Android permission scheme is represented via entity-relationship model which it has been used to model RBAC. The formal model is taken from Wook Shin et al. who explained the whole permission model and its proof in the paper [202]. The three major entities of the Android permission scheme are as follows:

- $APPS$, the set of applications
- $COMPS$, the set of application components
- $PERMS$, the set of permissions

Component-based construction of an application and declared, used, and enforce permissions can be represented in the relationships among the entities.

- $COMPOSE \subseteq APPS \times COMPS$, an 1:N relationship that expresses the composition of applications.
- $composes : (cmp : COMPS) \rightarrow APPS$, the mapping of component cmp onto its parent application.
- $DECLARE \subseteq APPS \times PERMS$, an 1:N relationship that maps an application to a set of permissions declared by the application.
- $declaredBy : (p : PERMS) \rightarrow APPS$, the mapping of permission p to an application that declares the permission.
- $USE \subseteq APPS \times PERMS$, a N:M relationship that depicts permissions used by applications.

- $uses : (app : APPS) \rightarrow 2^{PERMS}$, the mapping of application *app* to a set of permissions that *app* uses.
- $AENFORCE \subseteq APPS \times PERMS$, a N:M relationship that illustrates the permissions that are enforced by applications.
- $aEnforces : (app : APP) \rightarrow 2^{PERMS}$, the mapping of application *app* to a set of permissions that *app* enforces.
- $CENFORCE \subseteq COMPS \times PERMS$, a N:M relationship that illustrates the permissions that are enforced by application components.
- $cEnforces : (cmp : COMPS) \rightarrow 2^{PERMS}$, the mapping of component *cmp* to a set of permissions that *cmp* enforces.

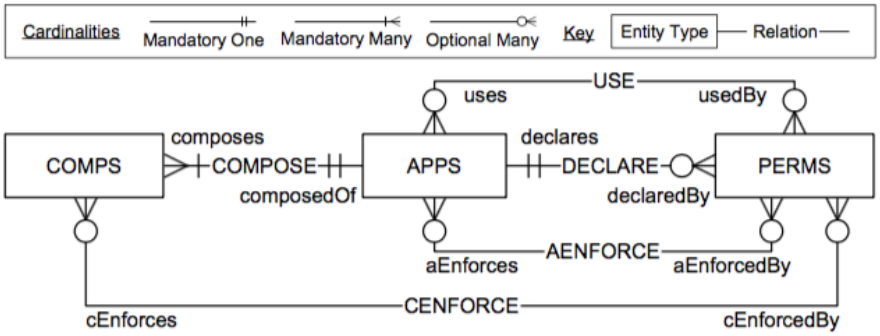

Figure 4.3: Entities and relations in the Android permission scheme [202]

Figure 4.3 shows the entities and the relations in the form of an Entity-Relationship diagram. The types and meanings of those figures not listed can be easily inferred. The following is a more detailed explanation of the cardinality constraints in the diagram. An application is composed of one or more components, and the components are introduced into a system as the application is installed on the system. Therefore, in the $COMPOSE$ relation, each element composes one application, and the application is $composedOf$ one or more components. An application declares some permissions or none. Declared permissions by an application are introduced into a system as the application is installed. Therefore, in $DECLARE$, an application optionally declares multiple permissions, while each permission has to be *declaredBy* an application. The relation USE, $AENFORCE$, and $CENFORCE$ are optional N-to-M relationships. An application can use or enforce some permissions. A component can enforce some permissions, as well. Conversely, permission can be used or executed by an application, or enforced by an element. None of the use or enforcement relations is mandatory.

The execution of a privileged operation on a protected resource object may require a permit. Therefore, the notion of permission is related to both operations and objects. The relations determine what kind of permission is required and when the permission needs to be checked. Figure 4.4 depicts the relations between permissions, operations, and objects. Note that applications and components correspond to objects in the figure, then the specification of the interactions between them follows. The additional entity and relation need to be added:

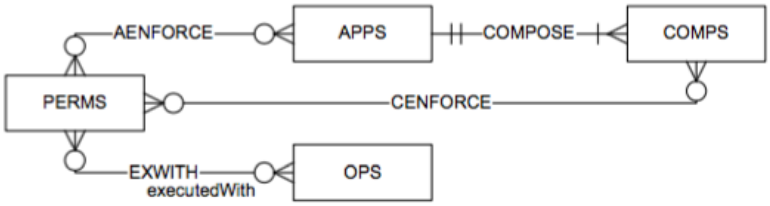

Figure 4.4: Android permissions [202]

- OPS, the set of operations.
- $EXWITH \subseteq OPS \times PERMS$, a N:M relationship that describes the permissions enforced on operations.
- $executeWith : (op : OPS) \rightarrow 2^{PERMS}$, the mapping of operation op to a set of permissions.

The definition of a permission includes the relation with a set of operations and the relation with a set of objects. However $AENFORCE$ and $CENFORCE$ can be collected from the manifest information, the former, $EXWITH$, can be obtained from concrete implementation, source codes of the Android framework and applications. For instance, when an activity component sets up its enforce-permission in the manifest file, it does not mean the permission is always checked whichever action is performed on the activity. When it gets started, the permission checking routines embedded in relevant API calls are triggered. Android's documentation guides the execution of which operation leads to the permission test. The relationship with operations can also be made when an application explicitly invokes check permission functions in its code.

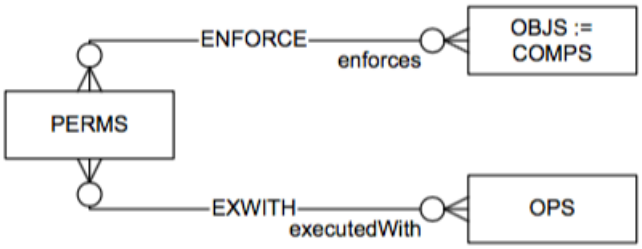

Figure 4.5: Android permissions (refined) [202]

The relations in figure 4.4 can be refined and redrawn in terms of components, as shown in figure 4.5. The new optional N:M relation, $ENFORCE$ reflects $CENFORCE$ when a component enforces permissions, or $AENFORCE$ when a component does not enforce permissions by itself. The mapping $enforce : (cmp : COMPS) \rightarrow 2^{PERMS}$ maps a component cmp to the set of permissions obtained by expression 4.1.

$$\{p : PERMS| \tag{4.1}$$
$$(cEnforces(cmp) = \emptyset \Rightarrow p \in cEnforces(cmp)) \vee$$
$$(cEnforces(cmp) \neq \emptyset \Rightarrow p \in aEnforces(composes(cmp))\}$$

The interaction between components is composed of operations that one element performs on the other. When the user calls one performer *scmp*, the other *ocmp*, and the sort of operation *op*, it can then denote the interactive process as a tuple of $(scmp, ocmp, op)$. Some of the actions are protected by permissions. Let the procedure that guards the consent protected operations as *checkAccess* which has the type of $checkAccess : (scmp, ocmp : COMPS, op : OPS) \rightarrow BOOL$.

The *checkAccess* tests the legitimacy of the interactive operation by calculating if *scmp* owns all of the permissions that *ocmp* enforces regarding op. If so, it returns $TRUE$, otherwise $FALSE$.

The set of permissions that *ocmp* enforces on *op* can be calculated by expression 4.2.

$$\{p : PERMS | p \in enforces(ocmp) \wedge p \in executedWith(op)\} \tag{4.2}$$

The set of permissions that *scmp* uses can be obtained by using *composes*() and *uses*() functions.

$$\forall p : PERMS, p \in enforces(ocmp) \wedge p \in executedWith(op) \tag{4.3}$$
$$\Rightarrow p \in uses(composes(scmp))$$

Finally, the definition of *checkAccess*() which returns $TRUE$ only if the condition expression 4.3 is satisfied.

4.4 Architecture Levels

According to Android architecture presented in figure 4.1 which is split into several levels, there is a similar distribution of security levels. The following section describes each of this security level from bottom to top. The core security principle of Android platform is that an adversary application should not harm the operating system resources, the user, and other applications. To procure the execution of this principle, the platform being a layered operating system exploits the provided security mechanisms of all the levels. Focusing on security, Android combines two level enforcement approaches: at the Linux kernel level and the application framework level (see figure 4.6).

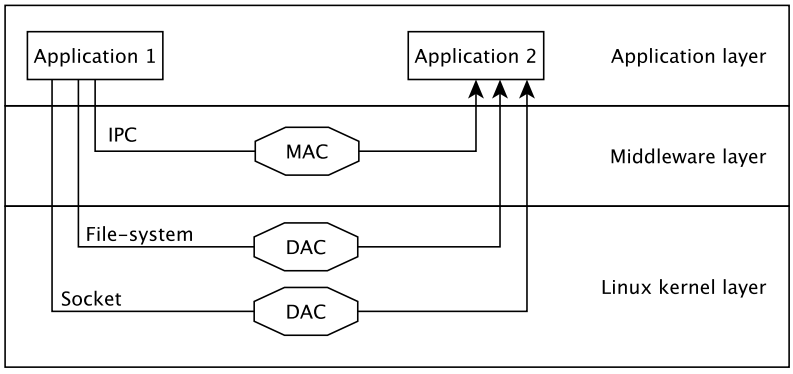

Figure 4.6: Levels of Android security enforcement [71]

The Linux kernel enforces the isolation of applications and operating system components exploiting standard Linux facilities [165] (process separation and DAC [139] over network socket and file system). This isolation is imposed by assigning each application a separate user ID and group identifier (GID) [198], as was discussed in section 4.2. Such architectural decision enforces running each application in a separate process. Thus, due to the process isolation implementation in Linux, by default applications cannot interfere each other and have limited access to the facilities provided by the operating system.

Therefore, application sandbox ensures that an application can not drain the operating system resources and can not interact with other applications [230]. The enforcement mechanism provided at the kernel layer efficiently sandboxes a request from other applications and the system component. At the same time, an active communicating protocol is required to allow developers to reuse application components and interact with the operating system units. This contract is called inter-process communication (IPC) [86] because it facilitates the interactions between different processes. In the case of Android, this protocol is implemented as the middleware between two architecture levels (see figure 4.6) with a particular driver released at the kernel level. The security on this level is provided by the IPC reference monitor [74]. The reference monitor mediates all communication between processes and controls how the applications access the components of the system and other applications. In Android, IPC reference monitor follows MAC principle [144].

Each application by default is run in a low-privileged application sandboxes. Thus, an application has access to a limited set of system capabilities. The operating system controls the access of applications to the system resources that may adversely impact user experience. This control is implemented in different forms, some of them are considered in details in the following parts of this section. There is also a subset of protected system features (e.g., camera, telephony or location functionality), the access to which should be provided to third-party applications. However, this access should be provided in a controlled manner. In case of Android, such control is realized using permissions. Each sensitive interface, which allows access to the protected system resources, is assigned with permission - a unique security label. Moreover, preserved features may also include components of other applications. To make the use of protected characteristics, the developer of an application must request the corresponding permissions in the file *AndroidManifest.xml* which is an inseparable part of each application.

During the installation process of an application, the operating system parses this file and presents the user a list of the permissions declared in this file. The installation of an application occurs according to all-or-nothing principle, meaning that the application is installed if all permissions are accepted. Otherwise, the application will no be installed at all. The permissions are granted just at the installation time, and there is a choice to allow or deny selected permissions in the settings manually. As an example of the permission format, consider an application that needs to send sms messages.

In this case, the *AndroidManifest.xml* file has to contain at least the tag depicted by the listing 4.1:

```
<uses-permission android:name="android.permission.SEND_SMS" />
```

Listing 4.1: SMS permission in *AndroidManifest.xml*

It is required to put the label with the specific meaning of the permission into tag *uses-permission*. An attempt of an application to use a feature, which permission has not

been declared in the *AndroidManifest.xml* file will typically result in a trowing of a security exception. Following sections are aimed into an introduction to the levels of architecture presented in figure 4.6.

Linux Kernel

In Android platform, Linux kernel [154] is responsible for process management, memory control, communication subsystem, file-system administration etc. While operating system mostly relies on the original version (“vanilla„) of Linux kernel functionality, several custom changes, which are required for the system operation, have been proposed to this level. Among them *binder* [195] - a driver, which provides the support for custom remote procedure call or inter-process communication mechanism on Android, *ashmem* - a replacement of the standard Linux shared memory functionality, *wakelocks* - a mechanism that prevents the system from going to sleep are the most notable ones [234].

Although these changes proved to be very useful in case of mobile operating systems, they are still out of the main branch of the official Linux Kernel.

One of the most widely known open-source projects, Linux has proved itself as a secure, trusted and stable piece of software being researched, attacked and patched by thousands of people all over the world. Not surprisingly, Linux kernel is the basis of the Android operating system. Android relies on Linux not only for process management, memory control, communication subsystem, file-system administration. It is also one of the most critical components of the Android security architecture. Linux kernel is responsible for provisioning application sandboxing and enforcement of some permission.

Application Sandbox

Let consider the process of an application installation in details. Applications are distributed in the form of *apk* package files. A package consists of a virtual machine executable resources, native libraries and a manifest file, and is signed by a developer signature. Three central mediators may install a package on a device in the stock operating system:

- Google Play
- Package installer
- Android debug brigde (ADB) install

Google play is a unique application that provides the user with a capability to look for a use uploaded to the application market by third-party developers along with a possibility to install it. Although it is also a third-party application, because of being signed with the same signature as the operating system, it has access to protected components of Android, which other third-party applications lack for. In case of the user installs applications from other sources he usually implicitly uses *Package installer* application. This system application provides an interface that is used to start a package installation process. The last named item *adb install* is the utility, which is provided by operating system, it is mainly used by third-party application developers. While the former two mediators require the user to agree with the list of permissions during the installation process, the latter installs an application quietly. That is why it is mainly used in developer tools aiming at installing an application on a device for testing. This process is shown in the upper part of figure 4.7.

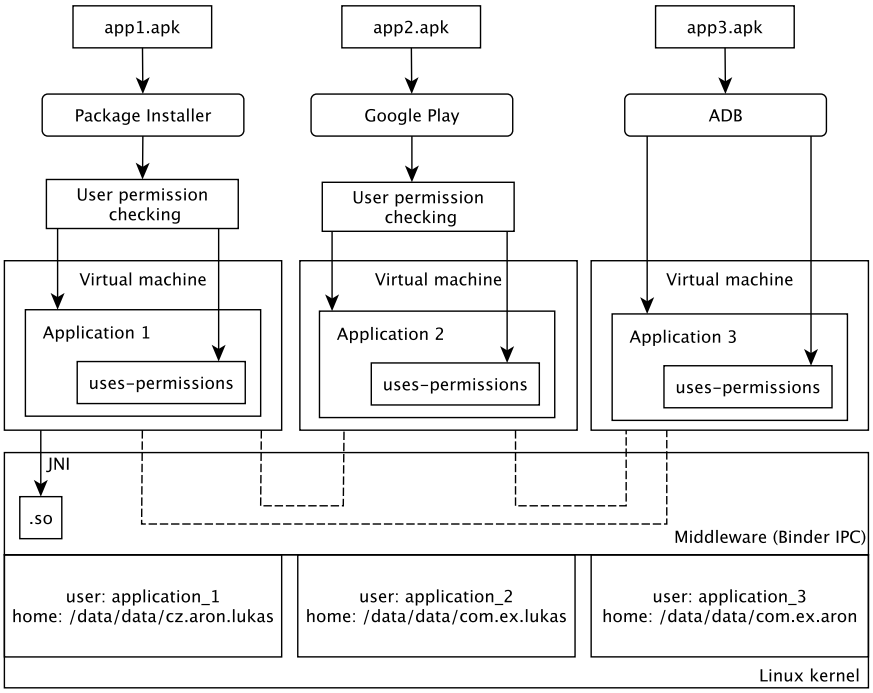

Figure 4.7: Android Security Architecture - Application installation possibilities [71]

Figure 4.7 shows a more detailed overview of the Android security architecture with the aim on application installation possibilities. The process of provisioning application sandbox at the Linux kernel level is the following.

During the installation process, each package is assigned a unique user identifier (UID) and a group identifier (GID) [198] that are not changed during application life on a mobile device. Thus, in Android, each application has a corresponding Linux user [117].

Username follows the format *app_x*, and UID of that user is equal to the value *Process.FIRST_APPLICATION_UID + x*, where *Process.FIRST_APPLICATION_UID* is the constant corresponds to constant with value 10 000.

For instance, in figure 4.7 *app1.apk* package receives during the installation process username with value application_1, and UID equal to 10 001. In Linux, all files in memory are subjects for discretionary access control (DAC). Access permissions are set by a developer of application or by an owner of a file for three types of users: the owner of the file, the users who are in the same group with the owner and all other users. This approach is the same as in the Linux based operating systems. For each type of users, a tuple of reading, write and execute permissions are assigned. Format is usually illustrated as *(r-w-x)* tuple.

Hence, so as each application has its UID and GID, Linux kernel enforces the application execution within its separate address space. Besides that, the application unique UIDs and GIDs are used by Linux kernel to enforce a clean separation of device resources such as memory, and CPU between different applications. Each application during the installation process also receives its home directory. Default path is usually set to the following target */data/data/package_name*, where the *package_name* is the name of an Android package, for example *cz.aron.lukas*.

Concerning Android, this folder is considered as internal storage, in which an application keeps its private data. Linux permissions assigned to this directory allows only the "owner„ application to manipulate the files in this directory, including the creation of new files. It should be mentioned there are some exceptions. The applications, which are signed with the same certificate, can share data between each other, may have the same UID or can even run in the same process. These architectural decisions set up effective and efficient application sandbox on top of Linux kernel level. This type of sandbox is base and straightforward on the verified discretionary access control model. Luckily, so as the sandbox is enforced on the Linux kernel level, native code and operating system applications are also subjects to there constraints described in this chapter.

Permission Enforcement

It is possible to restrict the access to some system capabilities by assigning the Linux user and group owners to the components that implement this functionality. This type of restrictions can be applied to system resources such as files, drivers, and sockets. Android uses file-system permissions and Android specific kernel patches known as Paranoid networking [71] to restrict the access to system features such as external storage, camera, and network sockets. Applying file-system permissions to files and device drivers it is possible to limit processes in accessing some functionality of a device. For instance, such technique is applied to restrict access to applications to a device camera. The permissions to */dev/cam* device driver is set to *0660*, with root owner and camera owner group. It means that only processes run as root or which are included in camera group, can read from and write to this device driver. Thus, only applications which are included into camera group can interact with the camera. The mappings between permission labels and corresponding groups are defined in the file *frameworks/base/data/etc/platform.xml*. The excerpt of the mapping file *platform.xml* is presented in the following listing 4.2.

```
...
<permissions>
...
        <permission name="android.permission.INTERNET" >
                <group gid="inet" />
        </permission>

        <permission name="android.permission.CAMERA" >
                <group gid="camera" />
        </permission>

        <permission name="android.permission.READ_LOGS" >
                <group gid="log" />
        </permission>
...
</permissions>
```

Listing 4.2: The mapping between permission labels and Linux groups

During the installation process are set the groups of the installed application. If an application request access to a camera feature and the user approve this request, the application is assigned to a camera Linux group GID (see the listing 4.2). Therefore, this

application receives a possibility to read information from */dev/cam* device driver. There are several points in Android where file-system permissions to files, drivers, and Unix sockets are set in: *init* process, *init.rc* configuration file, the *ueventd.rc* configuration file and system ROM file-system configuration file.

In traditional Linux based distributions, all processes are allowed to initiate a network connection. At the same time, for mobile operating systems the access to networking capabilities has to be controlled. In order to implement this control, proper kernel patches have been added that limit the access to network facilities only to the processes that belong to specific Linux groups or have specific Linux capabilities. These Android-specific patches of the Linux kernel have obtained the Paranoid name networking. For instance, for *AF_INET* socket address family, which is responsible for network communication, this check is performed in *kernel/net/ipv4/af_inet.c* file. The following listing 4.3 shows the part of the *af_inet.c* file, which is related to checking the access rights by the assignment to the specific Linux group. The following lines describe the function which calls the checking function to ensure the legitimate access to the requested feature.

```
...
#ifdef CONFIG_ANDROID_PARANOID_NETWORK
#include <linux/android_aid.h>

static inline int current_has_network(void)
{
        return in_egroup_p(AID_INET) || capable(CAP_NET_RAW);
}
#else

static inline int current_has_network(void)
{
        return 1;
}
#endif
...

/*
 *      Create an inet socket.
 */

static int inet_create(struct net *net, struct socket *sock,
        int protocol,int kern)
{
        ...

        if (!current_has_network())
                return -EACCES;

        ...
}
```

Listing 4.3: Paranoid networking patch (af_inet.c file)

The mapping between the Linux groups and permission labels fro Paranoid networking are also set in the *platform.xml* file, see listing 4.2. Similar patches are also applied to restrict the access to IPv6 protocol [63] and bluetooth [38]. The constants used in these checks are hard-coded in the kernel and specified in the *kernel/include/Linux/android_aid.h* header file, expressed in listing 4.4. Thus, at the Linux kernel level, the Android permissions are enforced by checking if an application is included into a special predefined group. Members of this group have access to the protected functionality. During the installation process of an application, when a user agreed all requested permissions, the application is included in the corresponding Linux groups and, hence, receives access to the protected functionality.

```
...
/* AIDs that the kernel treats differently */
/* was NET_BT_ADMIN */
#define AID_OBSOLETE_000 KGIDT_INIT(3001)
/* was NET_BT */
#define AID_OBSOLETE_001 KGIDT_INIT(3002)
#define AID_INET         KGIDT_INIT(3003)
#define AID_NET_RAW      KGIDT_INIT(3004)
#define AID_NET_ADMIN    KGIDT_INIT(3005)
/* read bandwidth statistics */
#define AID_NET_BW_STATS KGIDT_INIT(3006)
/* change bandwidth statistics accounting */
#define AID_NET_BW_ACCT  KGIDT_INIT(3007)
...
```

Listing 4.4: Hard-coded constants in the kernel level (android_aid.h file)

Native User-space

By the native user-space is understand as all user-space components that run outside of the virtual machine and do not belong to the Linux kernel layer. The first component of this layer is called - Hardware abstraction layer (HAL) [12]. HAL is blurred between the Linux kernel and native user-space layers. In Linux, drivers for hardware are either embedded into the kernel or loaded dynamically as modules to the kernel. Although Android is built on top of Linux kernel, it exploits a very different approach to support new hardware. Instead, for each type of hardware Android defines an API that is used by upper layers to interact with the current type of hardware.

The suppliers of hardware have to provide a software module that is responsible for the implementation of the API defined in Android for this particular type of equipment. Thus, this solution allows system not to embed all possible drivers into the kernel anymore and to disable the dynamic module loading kernel mechanism. Additionally, such architectural solution lets hardware suppliers select the license, under which their drivers are distributed.

Kernel finishes its booting procedure by starting one user-space process called *init* [234]. This process is responsible for starting all other processes and services, along with performing some operations in the operating system. For instance, if a critical service stops answering, the *init* process can reboot it. This process performs operations in accordance to the *init.rc* [234] configuration file. The toolbox includes essential binaries, which provide shell utility functionality in Android [234].

The operating system also relies on many vital daemons (long running background services). It starts them during system start-up and preserves them running when the system is working. For instance, *rild* - the radio interface layer daemon, which is responsible for communication between base-band processor and other systems, *service-manager* - contains an index of all *binder* services running in the system, *adbd* - Android debug bridge that serves as a connection manager between host and target equipment. The last but not least component in native user-space is set of native libraries. There are two types of native libraries: native libraries that come from external projects and developed within the platform itself. These libraries are loaded dynamically and provide various functionality for processes.

Booting Process

To understand what procedures provision security on the native user-space level, at first the booting sequence of the device should be considered. It should be mentioned that during the first steps this course may vary on different devices, but after the Linux kernel is loaded the routine is usually the same.

The flow of the booting process is shown in figure 4.8.

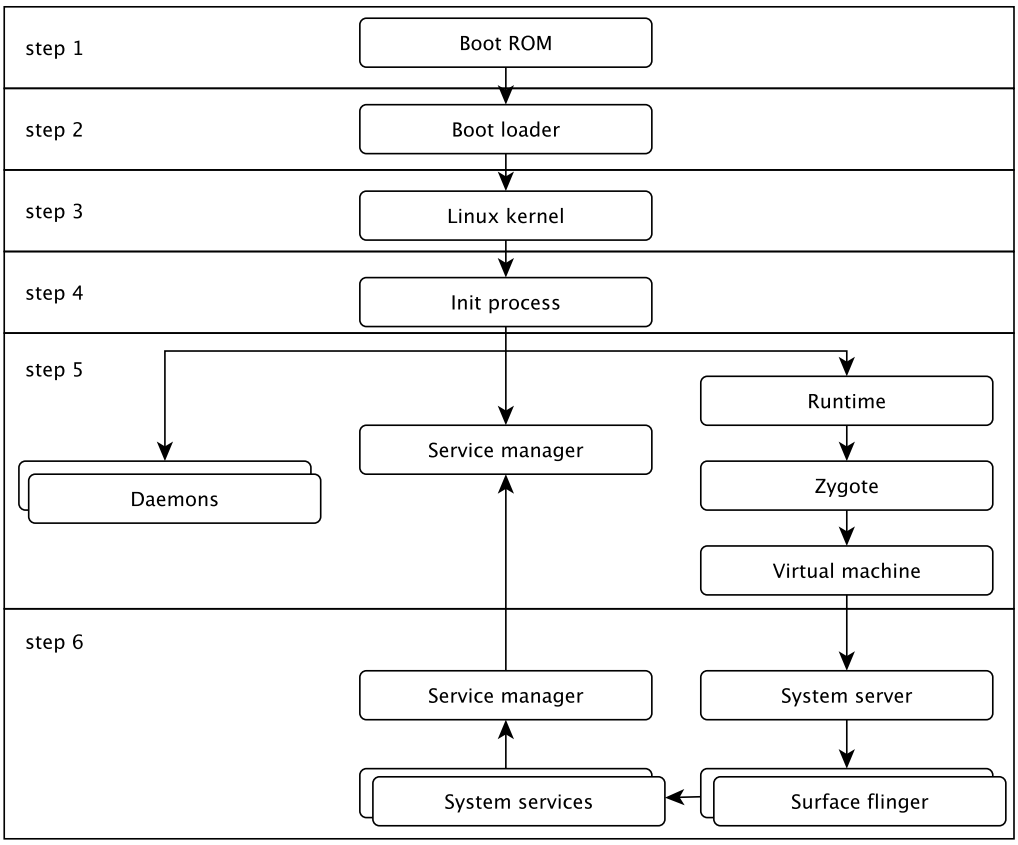

Figure 4.8: Android boot sequence

When a user powers on a mobile device then the CPU of the device will appear in a non-initialized state. In this case, a processor starts executing commands beginning from

a hard-wired address in the memory. This address usually points to a piece of code in the write-protected memory of the CPU, where boot ROM [237] is located. This routine is marked as *step 1* in figure 4.8. The main target of the code resided on boot ROM is to detect a media, where bootloader [173] is located. After the detection is done, boot ROM loads the bootloader into internal memory, which is available immediately after device powers on and performs a jump operation to the full code of the bootloader. The bootloader is illustrated as *step 2* in figure 4.8. The bootloader sets up external random access memeory, file-system, and network support. After that, it loads Linux kernel into the memory a passes the execution rights to it. Linux kernel initializes the environment to run another code (usually written in C language [121]), activates interruption controllers, sets up memory management units, defines scheduling, loads drivers and mounts root file-system. This step is illustrated in figure 4.8 with *step 3* label.

When memory management units are initialized, the system is ready to use virtual memory and run user-space processes [107]. Starting from this step, the process does not differ from the one that occurs on desktop computers running on Linux platform. The first user-space process, which is an ancestor of all processes in the Android operating system, is *init*. The executable of this program is located in the root directory of the file-system. To achieve additional settings of the *init* process there is configuration file *init.rc* which is written using a language called Android Init Language [204] and is also located in the root directory of the file-system. This configuration file can be imagined as a sequence of commands, which execution is triggered by the predefined events. The commands written in the *init.rc* configuration file defines global system variables, sets up basic kernel parameters for memory management configuration file-system. From the security, perspective is more important that it be also responsible for the basic file-system structure creation and the assignment of the owners and the file-system permissions to the created nodes.

Additionally, the *init* process is responsible for starting several essential daemons and processes. An executed process in Linux by default is run with the same permissions and with the same UID as an ancestor. In Android, *init* is started with the UID equals to zero, which means with the root privileges. Besides that, all descendant processes should run with the same UID. Fortunately, the privileged processes may change their UIDs to the less privileged ones. Thus, all descendants of the *init* process may use this functionality specifying the UID and the GID of a forked process. The owner and group id are also defined in the *init.rc* configuration file. The whole creation of *init* process is inside *step 4* in figure 4.8.

Another core process launched by this *init* process is called *Zygote* [12]. A Zygote is a special process that has been warmed-up. It means that the process has been initialized and linked against the core libraries. A Zygote is an ancestor for all processes (except the *init* process). When a new application is started, the request for the new process is handled by the Zygote. The Zygote process is forked and after that, the parameters corresponding to a new application such as UID, GIDs, and name, are set for the forked child process. The acceleration of a new process creation is achieved because there is no need to copy core libraries into the new process. The memory of a new process has a principle “copy-on-write„ [75] protection, meaning that the data will be copied from the Zygote process to a new one only if the latter tries to write into the protected memory. Core libraries cannot be changed. They are retained only in one place reducing memory consumption and the application start-up time. The Zygote process is part of the booting sequence as and is shown in figure 4.8 inside *step 5*.

The first process, which is run using Zygote is *System Server* illustrated in figure 4.8 in *step 6*. This process, at first, runs native services, such as *Surface flinger* [113] and *Sensor service* [161]. After the services are initialized, a callback is invoked, which starts the remaining services. All these services are then registered with the *Service manager* [246].

File-system

However, Android operating system is based on top of Linux kernel. Its file-system hierarchy does not comply with file-system hierarchy standard [5] that defines the file-system layout of Unix-based systems [242]. Android and Linux based operating system have some directories in common, for instance */dev, /proc, /sys, /etc, /mnt*, etc. The purpose of these folders is the same as in Linux. Moreover, there are folders, such as */system, /data* and */cache*, which cannot be found in the Linux based systems. These folders are the core parts of Android platform.

During the build of the Android platform, three image files are created: *system.img, userdata.img* and *cache.img* [200]. These images provide the core functionality of the operating system and the ones that are flashed on a mobile device. In the course of booting the system, the *init* process mounts these images to the predefined mounting points, such as */system, /data* and */cache* correspondingly.

The partition */system* incorporates the entire Android operating system except for the Linux kernel, which itself is located inside the */boot* partition. This folder contains the sub-directories */system/bin* and */system/lib* that contain core native executable programs and shared libraries respectively. Additionally, this partition encompasses all system applications that are built with the system image. To achieve that the content of this partition cannot be changed at run-time, thus the image is mounted in the read mode only. Hence, */system* partition is mounted as read-only, it can not be used for storing any data. For this purposes, the separate partition */data* is allocated and is responsible for storing user data or information changing over the time. For instance, */data/app* directory contains all *apk* files of installed applications, while */data/data* folder encloses the home directories of the applications.

The */cache* partition is responsible for storing frequently accessed data and application components. Additionally, the operating system over-the-air updates [142] are also stored on the partition before being run. So as */system, /data* and */cache* folders are formed during the compilation of Android platform, the default rights and owners to the files and folders contained on these images have to be defined at compilation time (before compilation). It means that the users and groups (UIDs and GIDs) should be available during the compilation of this operating system images.

Native Executable Protection

Some native binary applications are assigned with *setuid* and *setgid* access rights without user notification. These binaries are usually part of the *system.img* partition, which was discussed in the previous section. The settings for files and folders is stored in the file */system/code/libcutils/fs_config.c* which is located on every Android device. The excerpt of this file is mentioned in listing 4.5. For instance, the *su* program has the rights set. This public utility allows a user to run another program with the specified UID and GID. In Linux based operating systems this functionality is usually used to run applications with super-user privileges. According to listing 4.5 the binary */system/xbin/su* is assigned

with the access rights defined in number format (also called as flags) as is the definition of access rights on the Linux based systems. There is also another configuration file on every Android-based device, which is located in */android_filesystem_config.h*. This file contains the settings of ownership and groups to the daemons, sockets, system server, and hardware drivers.

```
static const struct fs_path_config android_files[] = {
    ...
    { 00644, AID_SYSTEM,    AID_SYSTEM,   0, "data/app/*" },
    { 00644, AID_MEDIA_RW, AID_MEDIA_RW,0, "data/media/*" },
    { 00644, AID_SYSTEM,    AID_SYSTEM,   0, "data/app-private/*" },
    { 00644, AID_APP,       AID_APP,      0, "data/data/*" },
    ...
    { 04750, AID_ROOT,      AID_SHELL,    0, "system/xbin/su" },
    { 06755, AID_ROOT,      AID_ROOT,     0, "system/xbin/librank" },
    { 06755, AID_ROOT,      AID_ROOT,     0, "system/xbin/procrank" },
    { 06755, AID_ROOT,      AID_ROOT,     0, "system/xbin/procmem" },
    { 04770, AID_ROOT,      AID_RADIO,    0, "system/bin/pppd-ril" },
...
}
```

Listing 4.5: Default permissions and owners (fs_config.c file)

Usually, in Linux, an executable application is run with same privileges as the process that has started it. The mentioned access flags allow a user to run a program with the privileges of an executable owner or group [52]. Thus, in this case, the binary */system/xbin/su* utility will be run as a root user. The root privileges allow the program to change its UID and GID to the ones specified by a user. After that, *su* may start the provided program with the specified UID and GID. Therefore, the program will be started with the required UID and GID.

In this case of privileged programs, it is required to restrict the circle of applications to have access to such utilities. In this regard, without such restrictions, any application may run this *su* program and obtain the root level privileges. In order to achieve this restriction on a native user-space level, there is implement the approach that compares UID of the calling program with the list of the UIDs allowed to run the restricted applications. Thus, the *su* executable application obtains the current UID of the process, which is equal to the UID of the process calling it, and it compares this UID with the predefined list of allowed UIDs.

Therefore, only if the UID of calling process is equal to constant *AID ROOT* or *AID SHELL*, then the required *su* utility will be started. To perform such check, the list of UID constants is required. These constants can be found inside *android_filesystem_config.h* file. Additionally, starting version 4.3 of Android operating system, the core developers use a principle of capabilities for Linux kernel system [182]. It allows them to additionally restrict the privileges of the programs that are required to run with root privileges. For instance, in the considered case of the *su* utility, it is not required to have all privileges of the root user. For this utility, it is enough to have a possibility to change the current UID and GID. Therefore, this program requires only *CAP_SETUID* and *CAP_SETGID* root capabilities to operate correctly.

Application Framework

Dalvik/Art is a registry-based virtual machine, and it allows the operating system to execute Android applications, which are written using Java language. During the build process, Java classes are compiled into a *dex* files that are interpreted by the virtual machine. The implementation of the virtual machine was specifically designed to be run in constrained environments. Additionally, the virtual machine provides functionality to interact with the rest of the system, including native binaries and libraries.

To accelerate the process of initialization, Android exploits a specific component called *zygote*[205]. It is a particular "pre-warmed„ process that has all core libraries linked in. When a new application is about to run, Android forks a new process from *zygote* and sets the parameters of the process according to the specification of the launched application. This approach allows the operating system no to copy linked libraries into a new process, thus, speeding up application launching operation. Java core libraries, which are used in Android, are borrowed from Apache Harmony project [231].

System services are one of the most critical parts of the operating system. Android comes with some system services that provide underlying mobile operating system functionality to be used by application developers in their applications.

For instance, *PackageManagerService* [30] is responsible for managing packages within the operating system, which means installation, update, deletion, etc..

Using *JNI* - java native interface [91] interfaces system services can interact with the daemons, toolbox binaries and native libraries of the native user-space layer. The public API to system services is provided via Android framework libraries. This API is used by application developers to interact with system services.

Binder Framework

As was described in previous chapters, all applications are run inside application sandbox. The sandboxing of the applications is provisioned by running all applications in different processes with different Linux identities. Additionally, system services are also run in separate processes with more privileged identities that allow them to get access to different parts of the system protected using Linux kernel DAC capabilities [193]. Therefore, an inter-process communication (IPC) [3] framework is required to organize data and signals exchange between different processes. In Android, a special framework called *binder* [3, 195] is used for inter-process communication. The standard POSIX system V [137] IPC framework is not supported by the Android implementation of the Bionic *libc* library. Moreover, additionally to the binder framework for some special cases Unix domain sockets [209] are used for communication with the *Zygote* daemon.

The binder framework was explicitly developed to be used in the Android operating system. It provides the capabilities required to organize all types of communication between processes in this system. Even the mechanisms, such as intents and content providers, well-known to application developers, are built on top of the binder framework. This framework provides the variety of features, such as the possibility to invoke the methods on remote objects as if they were local, synchronous and asynchronous method invocation, ability to send file descriptors across processes [195].

The communication between the processes is organized according to synchronous client-server model [41]. The client initiates a connection and waits for a reply from the server side. Thus, the communication between the client and the server can be imagined as they are executed in the same process thread. It provides a developer with the possibility to

invoke methods on remote objects as they were local. The communication model through binder is presented in figure 4.9.

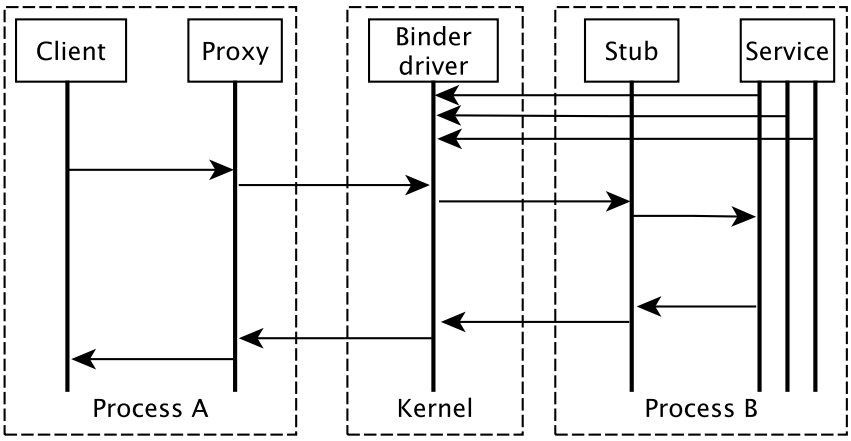

Figure 4.9: Binder communication model

In figure 4.9, the application in *process A*, which acts as a client, wants to use the behavior exposed by a service, which runs in the *process B*. All communication between clients and services using the binder framework happens through the Linux kernel driver */dev/binder*. The permissions to this device driver are set to world readable and writable. Hence, any application can write to and read from this device. To conceal the peculiarities of the binder communication protocol, the *libbinder* [188] library is used in Android platform. It provides the facilities to make the process of interaction with the kernel drive transparent for an application developer. In particular, all communications between a client and a server happen through proxies on the client side and stubs on the server side. The proxies and the stubs are responsible for marshaling [119] and unmarshaling [119] the data and the commands sent to the binder driver.

In order to make use of proxies and stubs, a developer just defines as Android interface definition language (AIDL) interface [157] that is transformed into a proxy and a stub during the compilation of the application. On the server side, a separate binder thread is invoked to process a client request. Technically, each service (sometimes called binder service) [167] is exposed using the binder mechanism and assigned with a token. The kernel driver ensures that this token represented as a numeric value (usually 32 bits long) is unique across all processes in the system. Therefore, this token is used as a handle to a binder service. Therefore, it is possible to interact with the service. However, to start using the service the client at first has to discover this token value. The discovery of service's handle occurs using binder's *context manager* [71]. The service manager is the implementation of binder's context manager on Android platform. The context manager is a special binder service with the predefined handle value equal to zero.

Whereas it has a fixed handle value, any part of the system can find it and call its methods. Context manager acts as a name service [54] providing the handle of a service using the name of this service. In order to achieve this behavior, each service has to be registered within context manager. For instance, the service can use a specific method of the *ServiceManager* class. Thus, a client has to know only the name of a service which it needs to communicate with. Using context manager the client receives the token which

is later used for the interactions with the required service. The binder driver allows only a single context manager to be registered in the system. Therefore, the service manager is one of the first services started during booting sequence. The components of service manager ensure that only the privileged system identities are allowed to register services. The binder framework does not impose any security by itself. At the same time, it provides the facilities to procure the security in Android.

The binder driver adds the UID and the PID of the sender process to each transaction automatically. Therefore, each application in the system has its UID. Then this value can be used to identify the calling party. The receiver of the call can check the obtained values and decide if the transaction should be completed. The receiver can get the UID, and the PID of the sender using the specific method calls [3]. Additionally, a binder handle can also behave as a security token due to its uniqueness across all the processes and the obscurity of its value [4].

Permissions

As considered in previous chapters, in Android each application by default obtains its UID and GID system identities. Additionally, there are also a number of the identities hard-coded in the operating system. These identities are used to separate the components of the Android operating system using DAC enforced on top of the Linux kernel level, thus increases the overall security of the operating system. Among these identities, *AID SYSTEM* stands out. This UID is used to run the *System server*, the component that unites the services provided by the Android system.

The *System server* has privileged access to the operating system resources, and each service runs within the *System server*. It provides the controlled access to a particular functionality to other system components and applications. The controlled access is backed by the permission management. It is connected to binder framework which provides the ability to get the UID and the PID of the sender on the receiver side. In general case, this functionality can be exploited by a service to control consumers that want to connect to the service. It can be achieved by comparing the UID and or PID of a consumer with the list of UIDs allowed by the service. However, in Android, this functionality is implemented in a slightly different manner. Each critical method of a service is guarded with a particular label called permission. Before running the method a check if the calling process is assigned with the required permission is performed. If the calling process has the required permission, then the service invocation will be allowed. Otherwise, a security check exception will be thrown.

For instance, if a developer wants to provide its application with a possibility to send sms message it is required to add the specific record (uses-permission) into application's *AndroidMainfest.xml* file, these user-permissions were depicted in listing 4.1. Android also provides a set of individual calls that allow checking at run-time if a service consumer has been assigned with permission. The permission model described so far provides an efficient way to enforce security. At the same time, this model is ineffective because it considers all the permission as equal. In the case of mobile operating systems, the provided capabilities may not always be equal in the security sense. For instance, the capability to install applications is more critical than the ability to send sms messages, which in turn is more dangerous than the setting an alarm or vibrating of the device.

This issue is addressed in Android by introducing the security levels of permissions. There are currently four possible levels of permission: *normal, dangerous, signature* and

signature-or-system. The level of permissions is either hard-coded into the Android operating system (for system permissions) or assigned by a developer of a third-party application in the declaration of a custom permission (inside *AndroidManifest.xml* file). To be granted, the *normal* permission has to be just requested in application's manifest file. The *dangerous* permissions, besides to be requested in the manifest file, have to be also approved by a user during installation or upgrade process. In this case, during the installation of an application, the user is displayed with the set of permissions requested by the package. If the user approves them, then the application will be installed. Otherwise, the installation is aborted.

The *signature* permission is granted by the system if the application requested the permission be signed with the same signatures as the application that has declared it. The details about usage of application signatures are considered in the following section. The *signature-or-system* permission is granted either if the application is requesting and the declaring the permission are signed with the same certificate or the requesting application is located on the system image. For instance, the vibrating capability will be protected with the permission of the *normal* level, send sms messages functionality will be guarded with the *dangerous* permission level and package installation ability will be secured with the *signature-or-system* permission level.

System Permission Definition

System permissions, which are used to protect Android operating system functionality, are defined in the framework's *AndroidManifest.xml* file located in *frameworks/base/core/res* folder of the Android sources. An excerpt of this file with several permission definition examples is shown in listing 4.6.

```
<manifest
  xmlns:android="http://schemas.android.com/apk/res/android"
  package="android" coreApp="true"
  android:sharedUserId="android.uid.system"
  android:sharedUserLabel="@string/android_system_label">
...
<!-- Allows access to the vibrator.
  <p>Protection level: normal
-->
  <permission android:name="android.permission.VIBRATE"
    android:label="@string/permlab_vibrate"
    android:description="@string/permdesc_vibrate"
    android:protectionLevel="normal" />
...
<!-- Allows an application to send SMS messages.
  <p>Protection level: dangerous
-->
  <permission android:name="android.permission.SEND_SMS"
    android:permissionGroup="android.permission-group.SMS"
    android:label="@string/permlab_sendSms"
    android:description="@string/permdesc_sendSms"
    android:permissionFlags="costsMoney"
    android:protectionLevel="dangerous" />
...
<!-- @SystemApi Allows an application to install packages.
```

```
    <p>Not for use by third-party applications. -->
    <permission android:name="android.permission.INSTALL_PACKAGES"
      android:protectionLevel="signature|privileged" />
...
</manifest>
```

Listing 4.6: The definition of the system permissions

In these examples the permission declarations are shown used to protect vibrator of the device, sending sms messages and package installation functionality. By default, the developers of third-party applications do not have access to the functionality protected with system permissions of levels *signature* and *signature-or-system*. This behavior is ensured in the following approach. The application framework package is signed with the platform certificate. Thus, the applications requiring the functionality protected with the permissions of these levels have to be signed with the same platform certificate. However, the access to the private key of this certificate is available only to the builders of the operating system, usually hardware suppliers, or telecommunication operators.

Permission Management

The system service *PackageManagerService* [204] is responsible for the application management on Android platform. This service assists with the installation, uninstallation, and update of applications on the mobile device. Another important role of this service is permission management. It can be considered as a policy administration point. It stores the information that allows checking if an Android application is assigned with a particular permission. Additionally, during the installation and upgrade processes of application, it performs a bunch of checks to ensure that the integrity of permission model is not violated during these routines.

Moreover, it also acts as a policy decision point. The methods of this service are the last elements in the chain of the permission checks. It is not considered the operation of *PackageManagerService* here, but there are sources for details of this service [109, 204]. *PackageManagerService* stores all information related to permissions of third-party applications in the file */data/system/packages.xml*. This file is used as a persistent storage between the restart of the system. However, at run-time, all information about permissions is preserved in random access memory (RAM) allowing to increase the responsiveness of the system. This information is collected during the boot sequence (see figure 4.8) using data stored in the *packages.xml* file for third-party applications and through parsing system applications.

Applications

Android application is a software that runs on Android platform and provides most of the functionality available to the user. The stock of operating system is shipped with some built-in applications called system applications. They are usually provided by the developer of the operating system in the case of Android it is Google [222] and other types of system applications are provided by the suppliers of the mobile devices. These applications compiled as a part of Android open source project (AOSP) [6] built process. Moreover, the user may install user applications from numerous application markets to extend the basic functionality of the operating system.

Application Components

Applications on the Android platform are distributed in the form of package *apk* file. A package consists of virtual machine executable files, resource files, a manifest file and native libraries. The package has to be signed by the developer of the application. To sign the package developer usually use a self-signed certificate. Each application can be built from several components which are provided by the platform. In the case of Android, there are four component types: *activity, service, broadcast receiver* and *content provider*. The separation of an application into the components support the reuse of application parts between application and also division of the application into more logical parts.

- *Activity* is an element of a user interface. The activity usually represents a screen, which a user can interact with.
- *Service* is a background worker. The service can run indefinite time. The most famous example of a service is a media player, that plays music in the background while a user is working with the device.
- *Broadcast receiver* is a component of an application that receives messages and starts a workflow according to the obtained message.
- *Content provider* is the last component that provides the ability to store and retrieve data. it also permits to share a set of data with another application.

Hence, the Android applications consist of different components. There is no central entry point unlike any programming language, such as Java programs with the main method. Regards to missing the central point, all components need to be declared inside the *AndroidManifest.xml* file by the developer of an application. There is an exception for *broadcast receivers* because they can be defined dynamically inside Java code of the application. Example of the application's *AndroidManifest.xml* file is in listing 4.7. This application is composed of one activity.

```
<?xml version="1.0" encoding="UTF-8" />
<manifest
  xmlns:android="http://schemas.android.com/apk/res/android"
  package="cz.aron.lukas.testapp"
  android:versionCode="1"
  android:versionName="1.0">

  ...

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

  <application
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name">

      <activity android:name=".TestActivity"
        android:label="@string/app_name">
```

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

Listing 4.7: Example of *AndroidManifest.xml* file

The operating system provides a variety of methods to invoke the components of applications. A new activity is started by using one of these methods *startActivity* and *startActivityForResult*. Services are started through the method *startService*. In this case, called service invokes its method *onStart*. When a developer is going to establish a connection between a component and a service he invokes the method *bindService* and the *onBind* method is invoked in the called service.

Broadcast receivers are started when an application or system component send a special message using the methods *sendBroadcast, sendOrderedBroadcast* and *sendStickyBroadcast*. Content providers are invoked by the requests from the content resolver. All other component types are activated through intents. The intent is a special mean of communication-based on the binder framework, which was already described earlier.

Intents are passed into the methods that perform component invocation. The called component can be invoked by two different types of intent - an *explicit intent* or an *implicit intent*. For the first intent type, the developer realizes picking the functionality in the component of his application and calls the component using the component name data field of the explicit intent. The other approach is to invoke a component of any other application, in this case, he has to be sure that this application is installed on the system.

Basically, from the developer's point of view, there is no difference between interactions of components inside one application or among components of a different application. For the second intent type, the developer transfers the right to choose the appropriate component to the operating system. The intent object contains some information in its fields, such as *action, data* and *Category*. According to this information, using *Intent filters* the operating system chooses the proper component that may process the intent.

An intent filter defines the "template„ of intents the component can process. Furthermore, the same application can define an intent filter that will process intents from other components.

Permissions on The Application Level

Permissions are used for protecting access to the system resources. The developers of third-party applications can also use custom permissions to guard the access to the components of their applications.

These permissions have to be defined inside *AndroidManifest.xml* file, an example of custom permission is in listing 4.8.

```
<permission android:name="cz.aron.lukas.permission.mypermission"
  android:label="@string/mypermission_label"
  android:description="@string/mypermission_description"
  android:protectionLevel="dangerous"
/>
```

Listing 4.8: Example of custom permission (part of *AndroidManifest.xml* file)

The declaration of custom permission is similar to the one of the system permissions. To illustrate the usage of custom permissions let refer to figure 4.10.

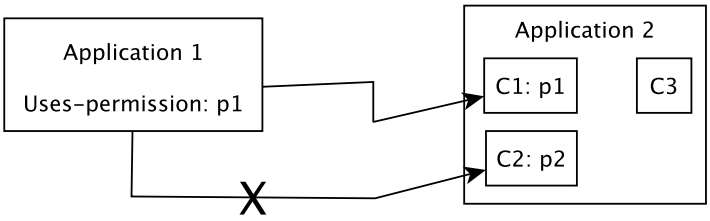

Figure 4.10: Permission enforcement to guard the components of third-party applications

The application 2 consisting of three components wants to protect access to two of them: *C1* and *C2*. To achieve this goal the developer of the application 2 has to declare two permission labels *p1, p2* and assign them to protect components correspondingly. If a developer of the application 1 wants to obtain access to the component *C1* of the application 2, it must be defined that the application requires permission *p1*. In this case, the application 1 receives a possibility to use the components *C1* and *C3* of the application 2. When the application has not specified the required permission, the access to the component guarded with this permission is prohibited, it is illustrated in figure 4.10.

For invoking of the components of the application is responsible *ActivityManagerService*. To enforce the security application components, in the framework methods, which are used to invoke the components, the special hooks are placed. These hooks check if an application has permission to call the required component.

These checks end in the *PackageManagerServer* class with the *checkUidPermission* method. Thus, the actual permission enforcement happens on the application framework level that is considered as a trusted part of the Android operating system. Hence the check cannot be bypassed by applications. More information about how the components are called and permission checks can be found in [228].

4.5 Summary

This chapter presented the architecture of one of the most used mobile operating system on mobile devices from the security point of view. The central part of this chapter was the description of platform layers used for protecting the user's data or privacy. There was the overview of security mechanisms, such as virtual machine, permission-based system and all layers of the architecture focused on security aspects of the platform. The most of this chapter was the detailed explanation of security aspects, related to the aim of this thesis, which is the protection user's data against data leakage. At this point, the data can be

sent outside the device within user actions, or by application logic through many paths. This chapter discusses these paths and also the permission models, which each application needs to follow.

Related to the open source of the platform, such as Android is, the snippets of a real source code of implementation were presented for the specific sections. The detail description of source code was aimed into arts that can influence the resulting solution of a prototype. Also, the operating system should not be modified the prototype needs to follow presented security properties and rules defined by the vendor.

The mobile operating system is the cornerstone for this thesis and can be considered as technical background for the implementation solution of a prototype. The next chapter is focused on a presentation the idea of the work and the definition of the concept.

Chapter 5

Definition of Access Rights Model

This chapter describes the main idea of this thesis which protects a user against leakage of privacy data from a mobile device. The core of this thesis is the formal definition of the novel approach to protecting data against leakage, but remain the functionality of the mobile device. The first step is to introduce the whole concept less formally and then describe the formal model.

The main concept is built upon the BYOD principle [163]. It means that a mobile device can be used as a personal device and also as a work device at the same time. The main issue with this principle is the security aspect. Moreover, the information which could be sensitive or corporate is taken outside the protected environment or company. The weakest point is the user and its device which is used for both purposes (personal and also work device).

Current solutions related to this area are mainly focused on two approaches. The first one is that the device is entirely administrated by the company - MDM principle [183]. The second one is supported by the manufacturer of the device. Manufacturer provides the decision at the boot time which mode the user would like to use - secured or unsecured. This approach is not convenient to a user who needs to switch between two modes all the time when requested data from one mode in another one and vice-versa. When the user does not want to care about these modes and still require to apply BYOD principle, there is a possibility to use one of the taint mechanisms that work on the following information flow and mechanism which handle the access to the data according to few aspects, such as position or data classification. More information about these tainting mechanism is described in the following sections.

5.1 Concept

This thesis is aimed at the BYOD principle with the ability to dynamically change the permissions which are granted to the application during installation. These permissions, already described in the earlier chapter), are dynamically changed according to the input files which the user requires to handle the current application. At the first phase, the files on the mobile devices need to be grouped into at least two primary categories - *public* and *private*.

When a user needs to work with its files (lets marked them as public files) with a current application the application remains with the same permissions as it has from the installation phase. The dynamic changing of permissions appears while using the files marked as

private. The same application which was used for public files can be used for private files as well. During opening this private file, the application recognizes that the file is marked as private and dynamically change permissions. All permissions which allowed the leakage file content from the device are denied at the application level. If the user wants to open more files, the application can work in two separate modes depends on the current working file. The public file is also able to be shared with other components, applications or even outside the device. On the other side, the private file is not able to be shared at all.

For example, the user has application *Text editor* which can work with the classical text files. Moreover, the application can send the opened file via various services outside the device through email, blue-tooth or internet connection. If the user opens the file marked as public, the application remains the same behavior as it was developed for. However, during the opening of a private file, the application is not able to send the file outside the device, but the functionality with the file remains.

Many implementations can handle the concept of this idea with modification of the operating system, which is not a convenient solution for a massive user base of the operating system. The required solution should be considered as a layer between operating system and application layer. The implementation of prototype discusses the solution which can be done by this way with some limitations. All works related to this topics are not able to provide this behavior without modification of the operating system. The next section discusses the solution of the taint tracking principles with the focus on solution related to dynamically changing permissions of other researchers.

5.2 Related work

This chapter contains the related work in security area focusing on permission flow tracking, changes in permission models, and related topics to this thesis. Besides, there are also covered kernel and sandbox modifications which are related to permission enforcement or logging mechanism. Data flow or permission flow tracking systems exist in two basic categories - static analysis and dynamic analysis. Hence, both categories are covered in this chapter.

5.2.1 Detection of Privacy Sensitive Information

According to research [118] there are dynamic analysis and static analysis to approach of malware detection. There is a possibility that a malicious developer can make his application to circumvent the detection of malware. To solve these problems, they focused on detection method using log output which is dynamic analysis. Linux debugging utility named *strace* [126] monitors system calls used by an application in Android.

There is a method performing malware detection by analyzing system calls that are obtained using the *strace*. Behavior using services of the kernel can be detected by this method. However, there is a problem that system call is not issued in the behavior which does not use services of a kernel, and it is impossible to detect such behavior. They focused on the fact that when API that retrieves the *phoneID* (unique phone identifier) is invoked, it is processed with a remote procedure call. It was proposed a method logging the invocations under the API by inserting *Log.v* method.

The logging by this technique cannot be avoided even if modification of API is performed on the caller side. Therefore, it is impossible to circumvent the detection even if a developer has malicious intent. Because this proposed method is implemented within Android

framework layer, a malicious application developer cannot interfere with modifications in this layer. In the current paper, they implemented proposal method tentatively and ran the application which acquires *phoneID* on the Android emulator. As a consequence, they confirmed record of invocation behavior of the *phoneID* acquisition API empirically.

Proposal techniques can be used to grasp a behavior of an Android application. As a countermeasure for Android malware, an examination of an application using proposal method can be applied. For example, distribution of malware may be prevented if Android market vendors examine applications in advance.

5.2.2 Overview of Information Flow Tracking Techniques

This article [146] covers the overview of flow tracking techniques and its comparison. Taint analysis techniques are used for tracking the information flow and possible leakage on Android [72, 111, 194, 196, 238]. The taint analysis [146] is data flow analysis technique that is popularly used to track the flow of sensitive information. In the taint analysis sources and sinks of sensitive data are predefined. Taint sources are nothing but the sources of sensitive information. In the context of Android taint, sources can be account, email, contact, calendar, database, file, location log, phone state, sms/mms, settings and unique identifiers, such as IMEI. Whereas taint sinks are points, from which data can leak out of the system. Common taint sinks in Android are the internet, publicly accessible storage, and others. Memory card, an inter-process communication message, and sms transmission. Taint tracking discovers whether there is a route from source to sink. If source data reaches the sink, it is identified as instances of data leakage. There are two approaches for taint analysis:

- Dynamic taint analysis
- Static taint analysis

Many papers have been proposed based on dynamic taint analysis [196]. The dynamic analysis can monitor code as it is being executed. It can provide precise security analysis based upon runtime information as it only considers single execution at a time. This approach of dynamic taint analysis observes the flow of information between sources and sinks. Any data value, which is derived from taint source, is marked as tainted and other values are left untainted. Taint propagation policy determines the flow of tainted data as the program executes. Under tainting and over tainting of sensitive information can lead to false positives and false negatives [47]. The dynamic taint analysis has performance overheads on Android as real-time monitoring of applications is performed. It provides detailed information on the specific run, but cannot provide complete information of all possible execution path of the program. This approach has to be implemented on the actual Android device or virtual Android device for real-time processing.

The static taint analysis approach is also proposed by many types of research in their works [82, 89, 122, 245]. This static taint analysis approach tries to cover all possible execution paths of the program. The complete code is statically analyzed without the need for its execution, generally, control flow graph (CFG) of a program is created. The CFG is used to trace the flow of sensitive information from sources to sinks. Modern static taint analyzers convert programs code into some intermediate representation, which can be effectively processed to generate CFG and call graphs [82].

The static analysis takes more time to analyze the program than dynamic analysis as it processes complete code and all execution paths. However, it has no real-time performance overhead as processing is done statically before the code is being executed. The static taint analysis on Android can be performed by extracting android package of all installed applications and then processing them outside of the mobile device. Also, it can be done at application market level.

TaintDroid

According to the article which presented TaintDroid proposal [72], a system-wide dynamic taint tracking system for Android. It can track multiple sources and sinks simultaneously. Mobile device users are notified at the runtime when sensitive information leaves the system. TaintDroid modifies the Dalvik virtual machine of Android to introduce variable level taint tracking in it using shadow variables. Each variable size is doubled from 32 bits to 64 bits with modified stack format. These extra 32 bits are used to store a *taint tag*. The *taint tag* is a value used to identify the sensitive information (e.g., location, IMEI number). For tracking taint sources TaintDroid does variable level tracking for interpreted code, method level tracking for native code, message level tracking for inter-process communication and file level tracking for secondary storage files. Taint tags are added to taint sources, and when it reaches to taint sinks, these tags are processed to identify which information is being leaked through that sink. In the Android operating system, each application runs in its sandbox with its user id (already discussed earlier), Dalvik virtual machine instance and set of permissions assigned to it [168].

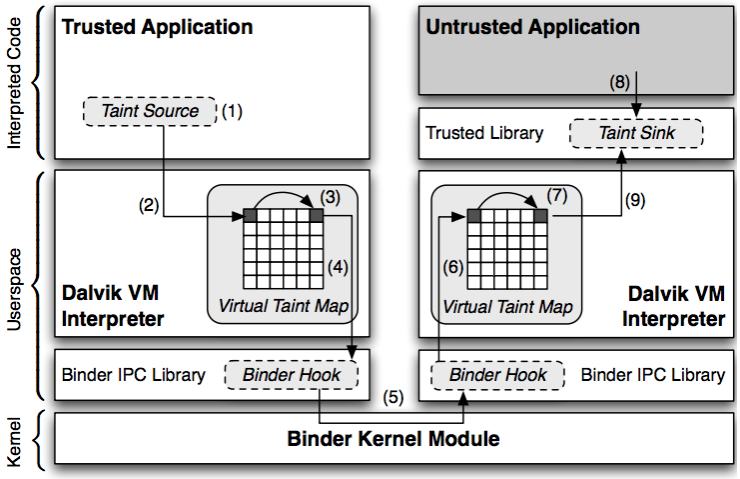

Figure 5.1: TaintDroid architecture within Android [72]

Figure 5.1 shows trusted and untrusted application running in their respective sandboxes on top of the kernel. Taint source is marked in the trusted application which is then mapped in the virtual taint map by the modified Dalvik virtual machine. The binder which is responsible for inter-process (inter-application) communication carries tainted data to the binder hook of untrusted application. The tainted data is then mapped and propagated in the corresponding virtual taint map of untrusted application related to the data flow rules. When the untrusted application invokes the taint sink specified library, tag from the tainted data is retrieved, and an event is reported to the user.

TaintDroid has a specific instruction set that requires a custom data-flow logic for taint propagation. Definition of taint markings, taint tags and variables follows, and taint propagation is presented in the following lines.

Definition 5.2.1. Universe of Taint Markings $\mathcal{L}$[72]

Let each taint marking be a label l. They assume a fixed set of taint markings in any particular system. Example privacy-based taint markings include location, phone number, and microphone input. They define the universe of taint markings $\mathcal{L}$ to be the set of taint markings considered relevant for an application of TaintDroid.

Definition 5.2.2. Taint Tag [72]

A taint tag is a set of taint markings. A taint tag t is in the power set of $\mathcal{L}$, denoted $2^{\mathcal{L}}$, which includes $\emptyset$. Each variable has an associated tag that is dynamically updated based on logic rules.

Definition 5.2.3. Variable [72]

A variable is an instance of one of the five variable types - method local variable, method argument, class static field, class instance field, and an array. Variable types have different representations. The local and argument variables correspond to virtual registers, denoted v_x. Class field variables are denoted as f_x to indicate a field variable with class index x. f_x alone indicates a static field. Instance fields require an instance object and are denoted $v_y(_f x)$, where v_y is the instance object reference variable. Finally, $v_x[.]$ denotes an array, where v_x is an array object reference variable.

Definition 5.2.4. Virtual Taint Map Function $\tau(.)$ [72]

Let v be a variable. $\tau(v)$ returns the taint tag t for variable v. $\tau(v)$ can also be used to assign a taint tag to a variable. Retrieval and assignment are distinguished by the position of $\tau(.)$ with respect to the $\leftarrow$ symbol. When $\tau(v)$ appears on the right-hand side of $\leftarrow$, $\tau(v)$ retrieves the taint tag for v. When $\tau(v)$ appears on the left-hand side, $\tau(v)$ assigns the taint tag for v. For example, $\tau(v_1) \leftarrow \tau(v_2)$ copies the taint tag from variable v_2 to v_1.

Definitions provide the primitives required to define run-time taint propagation for Dalvik VM. Table 5.1 captures the example of propagation logic, the extended version of propagation logic was presented in [72]. The table enumerates abstracted versions of the byte-code instructions specified in the Dalvik documentation [232]. Register variables and class fields are referenced by v_X and f_X, respectively. R and E are the return and exception variables, respectively, maintained within the interpreter. A, B, and C are constants in the byte-code.

Op Format	**Op Semantics**	**Taint Propagation**	**Description**
const v_A C	$v_A \leftarrow C$	$\tau(v_A) \leftarrow \emptyset$	Clear v_A taint
move v_A v_B	$v_A \leftarrow v_B$	$\tau(v_A) \leftarrow \tau(v_B)$	Set v_A taint to v_B
throw v_A	$E \leftarrow v_A$	$\tau(E) \leftarrow \tau(v_A)$	Set exception taint
sput v_A f_B	$f_B \leftarrow v_A$	$\tau(f_B) \leftarrow \tau(v_A)$	Set field f_B taint to v_A
sget v_A f_B	$v_A \leftarrow f_B$	$\tau(v_A) \leftarrow \tau(f_B)$	Set v_A taint to field f_B
iput v_A v_B f_C	$v_B(f_C) \leftarrow v_A$	$\tau(v_B(f_C)) \leftarrow \tau(v_A)$	Set field f_C taint to v_A

Table 5.1: Example of taint propagation logic [72]

The taint propagation logic uses conservative data-flow semantics for constant, move, arithmetic, and logic instructions. Destination register values are always entirely overwritten. Therefore the taint tag is set explicitly for each instruction. Constant values are

considered untainted and therefore do not contribute to the taint tag of the destination register. The interpreter maintains "hidden registers„ for return and exception values. These registers require taint tag storage and corresponding propagation logic. The arithmetic and logic operations include unary negation, binary arithmetic, bit shifts, and bit-wise AND and OR.

TaintDroid tracks information flows at real-time for privacy monitoring, and it has 14% performance overhead on a processor bound micro-benchmark. TaintDroid implementation needs building a custom ROM, i.e., patched version of the Android operating system requiring customized system release. TaintDroid has been integrated into CyanogenMod ROM [120] and the solution has been successfully released on Samsung Galaxy devices. Golam Swar [25] presented a collection of attackers on TaintDroid exploring its effectiveness and limitations. Here, the author successfully applied generic classes of anti-taint methods to circumvent TaintDroid.

AppFence

Peter Hornyack et al. [111] created *AppFence* tool to block sensitive information leakage using dynamic taint analysis approach. Instead of providing only notification to the user like TaintDroid, this tool blocks the application from sending sensitive data. AppFence could change the permission architecture of the Android operating system. It can provide to the user an option for restricting some permissions of the application. It implements two techniques data shadowing and exfiltration blocking to restrict the application from leaking sensitive information. The data shadowing substitutes shadow data in place of sensitive data to prevent it from exposure and exfiltration blocks network transmission that is carrying sensitive information. AppFence identified total eleven permissions referring to twelve essential sources of sensitive information for taint tracking.

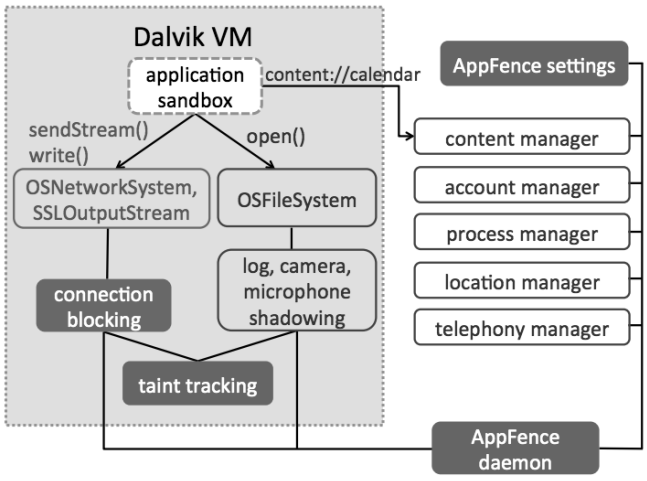

Figure 5.2: AppFence architecture [111]

The solid boxes in figure 5.2 are selected components introduced by AppFence in Android architecture for exfiltration blocking. The shadowing is done by modifying existing resource manager and file system components of Android. AppFence implementation is currently available for Android version 2.1 only, which is very old and unused version in these days. However this work can be marked as deprecated or obsolete, it still has the significant

contribution to the Android security and to taint the permissions with blocking operations. It has side effects on the application functionality, sometimes proper operations also get blocked and applications crash. This approach needs clear-cut differentiation between when to use shadowing and when to use exfiltration. It prevents the application from loading non-system native libraries. The application that can detect the presence of these security control may refuse main functionality until these controls are deactivated. In order to run this solution, there is the same requirement as in the TaintDroid proposal which means a special release of Android operating system with modified libraries and also other parts of the system to control the permissions flow.

Kynoid

Daniel Schreckling et al. [194] proposed Kynoid, which is real-time enforcement of fine-grained, user-defined and data-centric security policy. It is based on user-defined security policies defined for data items stored in shared resources. The core idea of Kynoid is to implement a middleware between application and the data as shown in figure 5.3 to provide policy enforcement functionality.

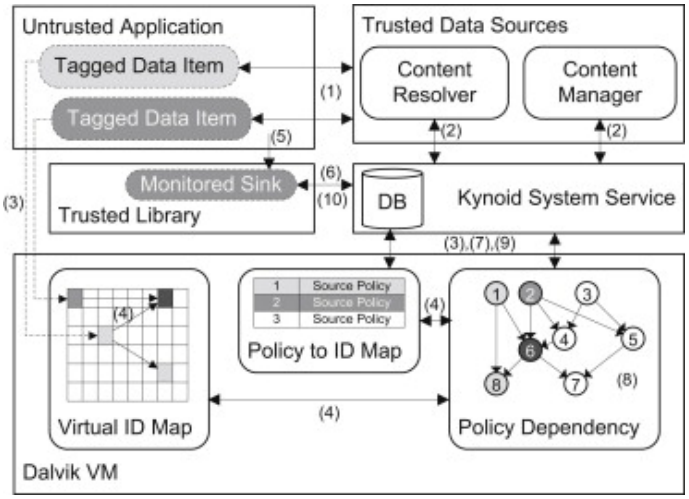

Figure 5.3: Kynoid architecture [194]

Kynoid is based on TaintDroid to integrate a lightweight policy tracker in sandboxing mechanism on Android platform. It tries to make the TaintDroid approach fine-grained to support efficient permission system which allows critical and non-critical data. TaintDroid supports 32 different tags in 32 bits field introduced in shadow variable, which can refer to at most 32 different data sources. Whereas Kynoid uses these 32 bits for identifiers, each variable in Android can be assigned a different identification number which is again mapped with a policy. It allows Kynoid to finer-grained tracking by having total 2^{32} mappings for security policies. However, this approach creates a tremendous amount of runtime and memory overhead which is addressed by using dependency graph in Kynoid.

The dependency graph is evaluated at the sink to derive exact security policy. Kynoid blocks the connections which are leaking information at monitored sinks as per policy defined. Implementation is done by modifying Dalvik virtual machine for taint tracking and Kynoid system service for policy database and identification numbers mapping. For inter-process policy tracking, identifiers of source variables are mapped to the identifiers of desti-

nation variable of another application. Sinks are monitored in similar kind of architecture that of TaintDroid to detect information leak. Kynoid claims to be giving a competitive performance on benchmark tests against TaintDroid while providing finer granularity of taint tracking policy, but it exists only as a prototype implementation. Also, Kynoid needs to analyze the impact of indirect flows to the overall performance.

LeakMiner

Zhemin Yang et al. [238] proposed LeakMiner, static taint analysis approach to scan applications on market site. As shown in figure 5.4, LeakMiner takes installation *apk* package file of the Android application and converts it to Java bytecode for additional processing. It also extracts the metadata from manifest file (part of the *apk* package file). Only reputable sources and sinks are considered in this approach for taint analysis. Permitted interfaces will be analyzed as Android only allows interfaces which are granted in permissions. The manifest file provides granted permissions for that application which is extracted as metadata. The call graph is generated from the transformed bytecode.

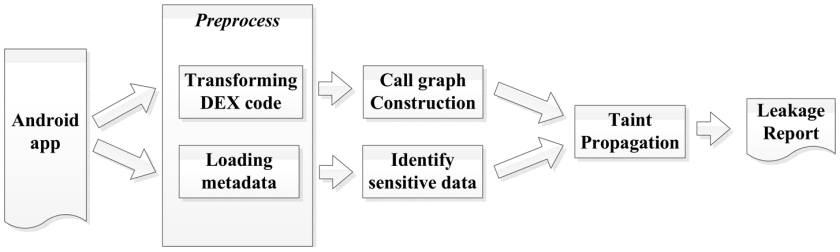

Figure 5.4: LeakMiner overall architecture [238]

To model Android activity lifecycle [112] callbacks and multiple entry points, different call graphs are linked to the root function node. In the taint propagation possible paths of sensitive data to taint sinks such as a network or local logging system are discovered and then reported to the user. Pointer analysis is applied to add some string context information to source and sink points. LeakMiner approach does not support implicit information flow leakage. It is not context sensitive which causes many false positives, precision is very less, about 50 %. On an average, each application takes approximately 2.5 minutes for analysis.

FlowDroid

Christian Fritz et al. [82] presented FlowDroid, highly precise taint analysis tool for Android applications. It takes the Android *apk* installation package file as input for processing and does the static taint analysis. This approach models entirely Android application lifecycle [112] precisely to handle callbacks. It is context sensitive as well as flow, field, and object-sensitive. Source and sink for targeted Android version are identified by using SuSi framework [22].

Detection of source, sink and entry-point is done by parsing manifest file, *dex* files and *xml* layout files extracted from the application *apk* installation file. FlowDroid generates main dummy method from parsed data as shown in figure 5.5. Taint analysis is performed on this graph to discover paths from source to sink. All the discovered paths are then reported to the user. IFDS framework [181] (interprocedural, finite, distributive, subset) is used to formulate interprocedural data flow analysis problem which creates exploded

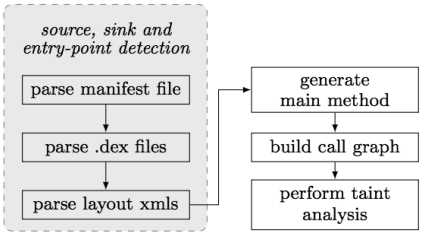

Figure 5.5: Overview of FlowDroid [82]

supergraph. FlowDroid ignores dynamic loading and reflection. It treats JNI [91] code as a black box and explicit taint propagation rules are defined for common native methods. It lacks taint tracking for intra-application and inter-application communication.

Also, the current implementation of FlowDroid ignores reflective calls and dynamically loaded codes. Along with FlowDroid, authors also proposed DroidBench [207] which is the very specific test suite for Android containing a set of vulnerable applications. On the DroidBench evaluation, FlowDroid outperformed other commercially available tools. It detected all seven data leaks of insecure bank application (the vulnerable application used for evaluation purposes). It also performed well on java specific benchmark suites. FlowDroid is highly precise static taint analysis tool, recently it is improved to support implicit flows, and it is available as an open source. It works on a computer where Android application *apk* files are given as input for analysis.

TrustDroid

TrustDroid presented by Zhibo Zhao et al. [245] addresses BYOD [163] privacy issues protecting leakage of corporate data. It can operate on the server (offline) as well as on the Android mobile device (real time). In offline mode, the static analysis is performed on applications that causes no performance issues for the mobile device. To operate efficiently in a realtime mode, it implements different levels of granularity. Static semantic analysis on the compiled bytecode is done for the data flow tracking. Taint propagation rules are defined for primitive data, object references, inter-process communication, native libraries and secondary storage.

Android bytecode is converted to the simple intermediate textual representation using Jasmin syntax format [159]. From the aspects of semantic analyzing Android bytecode is converted to the tree structure, which is again processed to generate call graph. This call graph is processed to discover source to sink paths. In order to generate the tree structure parser is built based on open source *ANTLR* parser generator [172]. TrustDroid taint tracking engine is composed of source and sink definition sets, file scanner, tag management system and an interface between these components. The engine scans output of semantic analyzer by using the file scanner and then performs the taint tracking.

TrustDroid can work as a standalone application on the mobile device with permissions to access the file system and scans *apk* installation files (which are available on the current mobile device). Limitations of this approach are inability do to analysis of dynamically loaded code and JNI [91] code. According to the author, this work can be extended further for the inter-process communication taint tracking. Authors have not included the results related to the performance against benchmarks of vulnerable applications. Also, it is not available as open source.

5.2.3 Summary of Related Work

All related work mentioned above is related to the tainting mechanism that has been chosen for this thesis. The main reason for this is the ability to improve mobile security without modification of the operating system. For this purpose is the mechanism of tainting with the mediation or inter-mediation of the process the certain way. These researchers are the cornerstone for the implementation of the prototype.

The implementation framework is not covered in this section but is mentioned in the chapter 6 and this framework is based on the knowledge introduced in this chapter. It is beyond this thesis to compare which solution is better against other, but there are mentioned the essential parts that could be used to prepare the proof of concept implementation - prototype in this case.

All related works have in common that are required root privilege access to the system to modify it. The modification is necessary for providing the required behavior which is taint mechanism. The question for this thesis is: is it possible to provide tainting principle without this administrator rights? The answer is in the following chapters.

5.3 Model of Required Behavior

This part informs about the required behavior defined formally to model the concept of the presented approach to protection. Since a user has a limited amount of files on the device there exists an easy way of modeling this behavior via the finite set of states and or finite sets of automata for each file separately. There are only two categories of files in the required solution. Thus the two types of automata are necessary or one with decision logic. Note that the decision logic is not considered as part of this work. The reason is that each user is individual and the categorization of files are different in various use cases.

Unfortunately, the model would not be considered as fulfilling required behavior in situations where a user changes a number of files on the device. This use case is a usual behavior of each mobile device user. Moreover, the formal definition should consider the almost unlimited amount of files available on a device. The formal definition is from preceding reasons split into two parts. The first part of the formal model is deterministic finite state machine (FSM) representing actions performed on the files from the user or application point of view. Automaton defined in figure 5.6 consists of five states, which defines the state of the file and also the decision of into which category the file belongs. The states have the following meaning: s - start, Spu - start public, Spr - start private, Cpu - close public, and Cpr - close private.

Figure 5.6 describes the automaton for file operations. The top branch is related to the public files, and the bottom branch is related to private files. This automaton is considered for the one specific file on the system. In order to cover all files in the system, this automaton needs to be defined independently for each file available on the device and also for new ones created in the future. The formal definition of this automaton is $FSM = (Q, \Sigma, \delta, s, F)$, where

- $Q = \{s, Spu, Spr, Cpu, Cpr\}$ - is a finite set of states.
- $\Sigma = \{open\_public, open\_private, share\_content, read\_public, read\_private, write\_public, write\_private, close\_public, close\_private\}$ - is a finite input alphabet.

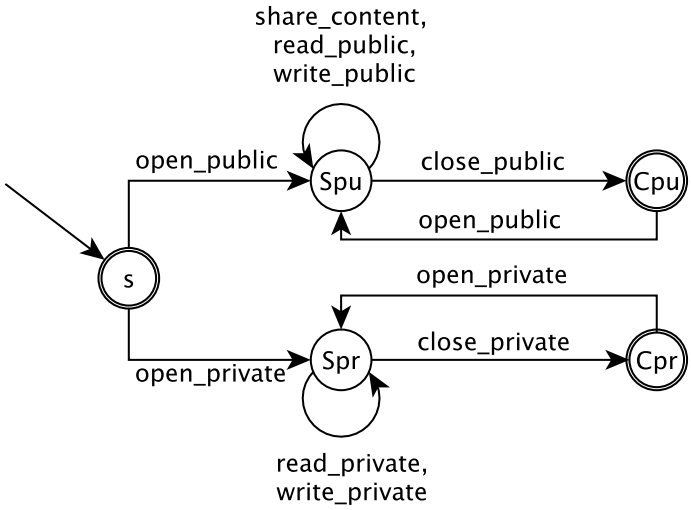

Figure 5.6: Required behavior on file level defined by finite state machine

- δ - is a state-transition function of type $Q \times \Sigma \rightarrow 2^Q$.
- $s \in Q$ - is an initial state
- $F \subseteq Q, F = \{s, Cpu, Cpr\}$ - is the set of final states.

States of the automaton define the working status of the file, and the transition between states identify the required behavior on file. The file can be opened as public or private. It is not allowed to work with one file with both approaches at one time. The file is marked as private or public, and the future changes are not considered in this model. For that purpose from a formal point of view, the history is required to repeatable working with the same file. It is depicted by the transition between states s into one of the states Spu, Spr and there is not possible to provide the transition back to the state s. Transition called *share_content* handle the availability of sharing the file outside of the device in all possible way. During this transition it is possible to send this file as an attachment to the email, share the file via any connection such as the internet, bluetooth, mobile network or through any other application feature.

Its initial state then defines the history of the file (after the original initial state s and open file operation). This decision logic should be defined by the classification logic which is not part of this thesis. When the decision is to make, then the file is defined as public or private the following operations are allowed.

Final states are marked with a double border in figure 5.6 and these states inform that the specific file was not open or it was successfully closed. The FSM is deterministic. Each state has exactly one transition for each possible input. The definition of state-transition function δ is defined in the table 5.2.

The second part of the formal model is the higher view over the possible transitions on file. As was already discussed, the FSM needs to be defined independently for each file available on the device. To model that behavior the automaton presented by Alan Turing [104] - Turing machine (TM) was chosen. In order to define the formal model by the TM which simulates the FSM (see figure 5.6), the logic needs to be defined.

$\delta(s, open\_public) = \{Spu\}$	$\delta(s, open\_private) = \{Spr\}$
$\delta(Spu, close\_public) = \{Cpu\}$	$\delta(Spr, close\_private) = \{Cpr\}$
$\delta(Cpu, open\_public) = \{Spu\}$	$\delta(Cpr, open\_private) = \{Spr\}$
$\delta(Spu, read\_public) = \{Spu\}$	$\delta(Spr, read\_private) = \{Spr\}$
$\delta(Spu, write\_public) = \{Spu\}$	$\delta(Spr, write\_private) = \{Spr\}$
$\delta(Spu, share\_content) = \{Spu\}$	

Table 5.2: Required definition of state-transition function (δ) for FSM.

For clear solution the TM has two tapes, the first one is input tape with the operation sequence flow of available files and the second tape (state tape) handle the state of the FSM for the specific file. The overview of designed TM is shown in figure 5.7.

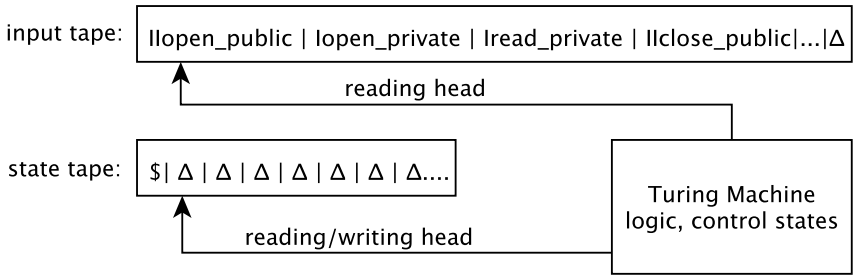

Figure 5.7: Example of Turing machine with two tapes

In order to have the connection between the operations on files and its states there needs to exist the transformation of each file into the unique sequence of one symbol. The definition of the transformation is depicted in definition 5.3.1.

Definition 5.3.1. Let $\mathbb{N}_+$ is a set of positive integer values, $F = \{F_1, F_2, \ldots, F_n\}$ be a set of files available on the mobile device, where n is a count of these files ($n \in \mathbb{N}_+$). There exist mapping function $transform$, that maps $\forall f \in F, \exists p \in \mathbb{N}_+ \Rightarrow transform(f) = p$. In other words, each element file $f \in F$ is paired with exactly one element of the set of positive integer values $p \in \mathbb{N}_+ \vee p > 0$ by the mapping function $transform$.

The positive integer can be expressed as the sequence of symbol I, which defines the power value of the element p.

For instance the positive integer value $p = 2, p \in \mathbb{N}_+$ is transformed into sequence of symbols I into the following string: II. According to the transformation mentioned above, each possible file operation has a prefix with the unique sequence of symbols I. This approach determines which operation is applied to the specific file. For instance operation *open_public* for file defined by unique sequence II has the format II*open_public*.

There is the connection between the file operation and the file on which the operation is applied. Operations on the files defined by this approach are the input alphabet on the first tape of the Turing machine.

The whole formal definition of Turing machine as a model is $TM = (Q, \Sigma, \Delta, \Gamma, \delta, s, F)$, where

- $Q = \{1, 2, 3, 4, A, R\}$ - is a finite set of states.

- Δ - is a blank symbol of the tape denoting the unused space on the input tape.
- $\Sigma \setminus \{\Delta\}$ - is the set of input symbols, that is, the set of symbols allowed to appear in the initial tape contents. This alphabet appears on the first tape only.
- $\Gamma = \{Spu, Spr, Cpu, Cpr\}$ - is a finite set of tape alphabet symbols which appear on the second tape only.
- $\delta : (Q \setminus F) \times \Sigma \bigcup \{*\} \times \Gamma \bigcup \{*\} \rightarrow Q \times \Gamma \bigcup \{L, R, \_\}$ - is a transition function, where $*$ is any symbol, L is left shift, R is right shift, and _ is no-operation symbol.
- $s \in Q, s = 1$ - is the initial state.
- $F \subseteq Q, F = \{A, R\}$ - is the set of final states.

According to using two tapes, the two reading heads are necessary to read values from both tapes at the same time. However, this operation can be simulated as a sequence of two atomic operations provided reading one symbol on the first tape and then reading one symbol on the second tape, it is more transparent to perform such operation via two reading heads. When reading or writing is omitted (no operation is provided at the specific moment) on one of the selected tapes of the TM the (_), symbol is used.

The allowed input symbols on input tape are the files represented in the form of sequences of symbol I, as was already defined, and the name of operations on these files. Legal input can be defined as regular expression 5.1. After reading the symbol from the input tape (first tape of TM), the reading head moves to the right automatically.

$$\Sigma = I^{+}[open\_public|open\_private|share\_content|read\_public|read\_private \\ |write\_public|write\_private|close\_public|close\_private] \tag{5.1}$$

This regular expression 5.1 defines all possible combination of files with available operations and thus defines the whole input alphabet Σ. The second alphabet Γ of TM which is related to the second tape is the same set from the set of states presented with FSM illustrated in figure 5.6. The reason is that on the second tape the TM simulates the FSM for each file. In more details, the input tape is the source of all possible operations on any files available on the device and the second type is the logic for each file already defined by the FSM.

The connection between the operation on the specific file and the second tape defines the sequence of the symbols I. This sequence defines the index of the cell where a state of FSM is saved on the second tape. For instance, the sequence on the first tape in the format $III open\_public$ operates with the third cell of the second tape.

From the definition of TM, the tape is bounded from the left side (both tapes). To handle the move to the leftmost position let define the specific symbol, such as ($) which is the first cell (with index equal to zero) of the second tape denoting that this is the leftmost cell which cannot be used. Its a boundary and the tape cells start after this symbol. Related to the movement to the leftmost cell the transition should be defined for each symbol which is not equal to the boundary symbol. Turing machine which models the required behavior is presented in figure 5.8.

Related to figure 5.8 and its transitions are defined as a tuple, where the first part is the reading from input tape and the second part denotes reading from the second tape.

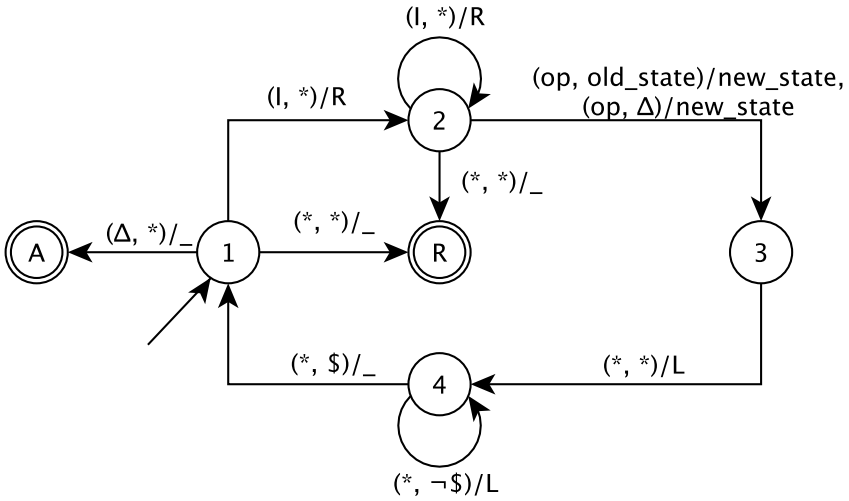

Figure 5.8: Turing machine models required behavior

The writing operation is defined after the (/) symbol, and the result is written onto second tape (or movement is performed). The transition follows the formal definition of δ. Also, there appears the symbol of $(*)$ which defines wild-card with the meaning that it does not matter on the symbol under the reading head. Note that the symbol of the star $(*)$ is used only if the more specific transition is not possible. This type of transition has the lowest priority because it is usually used to halt the TM.

For the clarity of the condition on the transition is defined in the form of expression $(\neg\$)$ denoting that until the symbol under the reading head is not equal to boundary symbol the reading head will continue with the specific operation. Specifically this expression is used in the state 4 in which the reading head is moved to the boundary of the state tape and the transition from state 4 into state 1 defines the starting position of the state tape.

The transition between states 2 and 3 is defined as operation (op) on the file with the required transformation in definition 5.3.1. The reading head on input tape read previous state (old_state) of FSM (or starting symbol Δ, on FSM defined as state s) from the second tape and provide operation of FSM with resulting new state (new_state) which is written to the second tape.

Note that its very important to clarify the difference between symbols (_) and $(*)$. The reading operation defined on the first tape (input tape) is usually composed of two primitive operations - reading and moving the reading head one position to the right. In turn, the two operating symbols have a different meaning. The star symbol $(*)$ means wild-card for any symbol available on the current tape, which does not cover the movement operation. It helps the logic with moving reading head on the second tape.

For instance, when the reading head on the second tape needs to move to the first position of the tape and the reading head on the first tape persists on the same position. Otherwise, the symbol (_) determines the operation that provides no action, sometimes called no-operation.

The last missing part of this formal model, which is the definition of transition function δ, is depicted in table 5.3.

$\delta(1, I, *) = \{(2, R)\}$	$\delta(1, \Delta, *) = \{(A, \_)\}$
$\delta(1, *, *) = \{(R, \_)\}$	$\delta(2, I, *) = \{(2, R)\}$
$\delta(3, *, *) = \{(4, L)\}$	$\delta(4, *, \neg\$) = \{(4, L)\}$
$\delta(2, open\_public, \Delta) = \{(3, Spu)\}$	$\delta(2, open\_private, \Delta) = \{(3, Spr)\}$
$\delta(2, open\_public, Cpu) = \{(3, Spu)\}$	$\delta(2, open\_private, Cpr) = \{(3, Spr)\}$
$\delta(2, close\_public, Spu) = \{(3, Cpu)\}$	$\delta(2, close\_private, Spr) = \{(3, Cpr)\}$
$\delta(2, share\_content, Spu) = \{(3, Spu)\}$	$\delta(4, *, \$) = \{(1, \_)\}$
$\delta(2, read\_public, Spu) = \{(3, Spu)\}$	$\delta(2, read\_private, Spr) = \{(3, Spr)\}$
$\delta(2, write\_public, Spu) = \{(3, Spu)\}$	$\delta(2, write\_private, Spr) = \{(3, Spr)\}$

Table 5.3: Required definition of state-transition function (δ) for TM.

As was described the automaton works with a specific format of input - a name of the operation with a prefix of any amount symbols I (at least one). During the reading of these symbols from input tape, the reading head on the state tape moves its head to the right with the same amount of movement as some symbols I. In this approach, the reading head of state type has the correct position of the cell which defines the state of the FSM for the specific file defined on the input tape. After the file follows the operation on file and this operation has to satisfy the required behavior previously defined by FSM or by transition function δ.

Automaton halts when there is the wrong symbol on input tape, unsupported transition within FSM, and or wrong file transformation. There are two finite states, the first one - A is state denoting acceptance of the input. The whole input tape needs to be read, and if the reading was successful according to file transformation and following all FSM transitions, the machine accepts operation sequences on files. Otherwise, the machine halts in the second finite state - R with the meaning that the input was not valid and the state of the whole TM is rejected.

5.4 Summary

This chapter introduced the main idea of this thesis. There were introduced other work of researchers in the similar areas with the focus on mobile devices. Related to other work that can provide similar protection, but not in a dynamical way as it is required, moreover there is not any paper describing the prototype which does not implement the required behavior without modification of the underlying operating system.

The most important part of this chapter is the definition of the idea that is the cornerstone of implementation solution. Therefore the model of required behavior was defined formally, and this model can be used for the verification purposes. The model of required behavior was split into two working parts, in which the first one (finite state machine) defines the behavior on specific file categories. The transitions of this automaton are described in the format of file operations. To be able to work with the dynamically changing environment (files can be removed or new ones created) the Turing machine simulates this finite state automaton on the second tape. A mathematical definition was presented in the form of Turing automaton that simulates the unlimited amount of files on the second tape, which represents the finite state automaton with required behavior. To determine which file is defined by which automaton the transformation function which maps file into sequence symbols I, were presented during this chapter as well.

As was already mentioned the next chapter discusses the implementation of the prototype, which should follow the model of required behavior. Therefore, the model of implementation solution is also part of the next chapter.

Chapter 6

Implementation of Prototype and its Model

In this part of the thesis is described an implementation of the proposed mediator between applications and the operating system. The implementation of prototype consists of the design of the solution, used a framework to handle mediation and last but not least the model of the implementation. Besides, this model should be in contrast with the model of required behavior, presented in the previous chapter.

However, there are many other works related to improving security specifically with the aim of taint tracking as was described earlier. There are still gaps which are not covered by other work. For example, the novel approach is made by changing permission enforcement according to open files during application. Moreover, this approach could be moved into another level not to limit the decision on file level only. The logic could be improved and defined on any other input of the application. It is related to taint the low-level system call allowance or denying automatically. It is what is covered in this chapter and precisely in the following sections.

6.1 Framework

The implementation framework for the prototype is called Aurasium [233]. The whole project has been developed since 2012 as an intern project at the University of Cambridge. The central philosophy is a mechanism of unpacking android application package - *apk* file, injection of monitoring code into the application and then put the modified files back to the package file. It does not require any administrator (root) access. In order to attach code, which runs inside the sandbox, the project exploits operating system architecture of mixed java and native code execution and introduces interposition code by *libc* library [123]. In order to mediate almost all types of interactions, this approach seems one of the best, because this library is the main point of interaction between the Android operating system and applications.

The framework is split into three main parts. As was already introduced, there has to be partly responsible for manipulating with a *apk* files. This automated repackaging system is called *pyAPKRewriter* and as the name advises the application is written in Python programming language [217]. Second and essential part of the Aurasium framework is the monitoring code included in *ApkMonitor* application that intercepts an application's

interactions with the system and the *Aurasium's Security Manager* application, which enabling convenient handling of policy decision of all repackaged application on the device.

The framework structure is shown in figure 6.1 and starting with the sandboxing code, the top layer of the framework is written in Java language [92]. The aim is to create a well-documented and easy-to-use abstraction layer on top of a cumbersome native layer of the framework. The lower layer provides interface for other possible programs and delegates all requests to the low-level part of the framework implemented in a native C++ language [210]. This layer consists of few shared objects that do all the magic work, such as communication with the virtual machine or establishing the mechanism for inter-process communication.

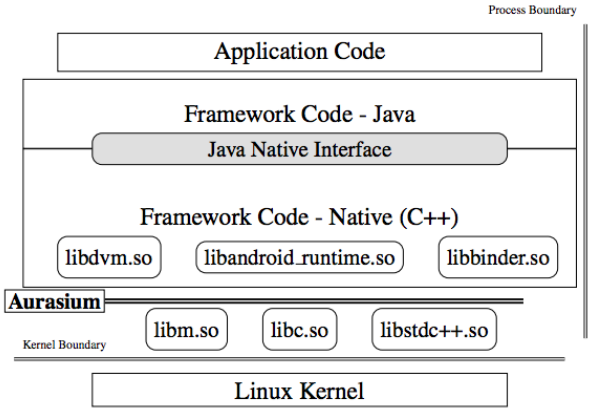

Figure 6.1: Aurasium framework structure [233]

The second part of Aurasium framework, the Python script for repackaging utilizes the previously mentioned sandboxing code and deploys it to the Android *apk* application installation package file. According to figure 6.2, besides the sandboxing code, Aurasium has to include also several additional parts to *apk* file in order to provide the functionality.

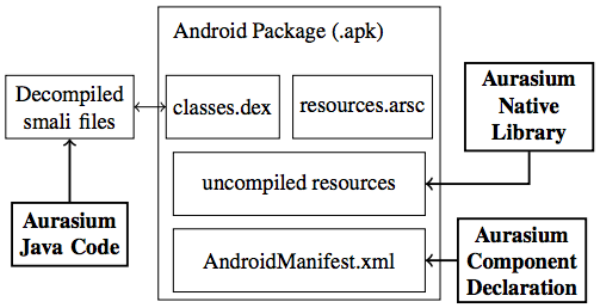

Figure 6.2: Aurasium repackaging system [233]

The last part of the framework is called Aurasium security manager (ASM) as was already introduced. ASM handles the policy decisions centrally, which means that all repackaged applications can be maintained at one place. Security policy is based on the decision of application or user. Application decision works transparently without user interaction, while the user decision is consented by a dialog window and can be remembered and use by default during next application run. Since the project is introduced and publicly doc-

umented in [233] only superficially, the more significant part of the information is hidden and have to be obtained by research or source code analysis.

Principle of Mediation

Aurasium mediation mechanism is based on the interposition code of Android's standard library called *libc* [123, 233]. This library is used when the upper layer of the framework wants to interact with the operating system. It is located directly on top of the Linux kernel and initiates appropriate system calls into the kernel that completes the required operation.

The library is directly mapped in logical address space (LAS) [44] of each process of every Android application using dynamic linking mechanism. Overview of the mapping into LAS is shown in figure 6.3. This approach is maintained by C++ linker *ld.so* [244], which interconnects arranged compile code of *libc* library with the framework libraries code in the LAS. In the Linux as well as in the Android operating system, each compiled library located in the LAS, and shared object files on the disk are in the executable and linking format (ELF) [240], thus the library could be shared and therefore mapped anywhere in the LAS. To achieve injection of the library into LAS the mechanism of position independent code (PIC) [197] has been used.

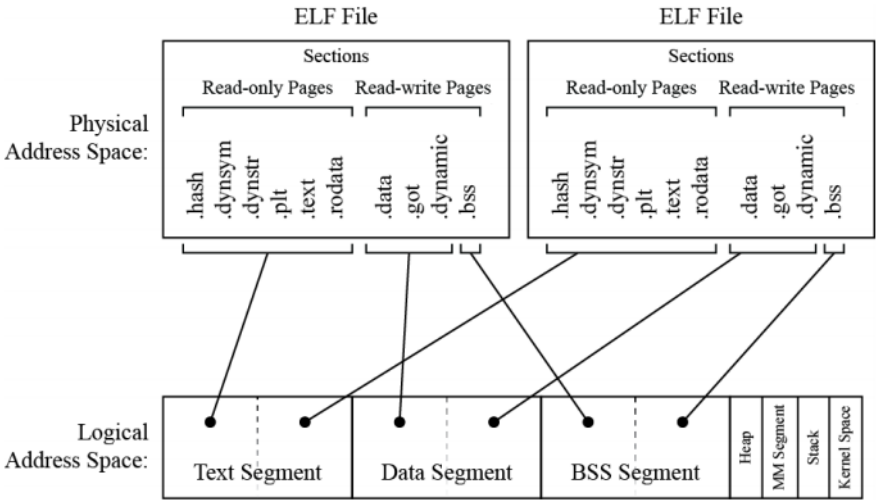

Figure 6.3: Mapping into process's logical address space [233]

For this purpose, ELF object file dispose dynamic symbol table (DST) [33] in *.dynsym* section containing all of the file's imported and exported symbols used by linker to fill the prepared read-write pages. In more details, *.got* and *.dynamic* sections are used - *.got* section contains global offset table [50] and is used by setup functions in procedure linkage table to retrieve the real target address of the remote function. Section with *.dynamic* is used to tag the values during linking process.

Since the global offset table is located at the fixed distance from the text segment, instructions in the code can jump to the correct offset entry even if the library has been mapped to the arbitrary address. Firstly, the linker collects and maps all the libraries code and data into the LAS of a process, and after that, it fills the global offset table with absolute addresses to ensure communication between modules and libraries.

Hence, Aurasium goes through every loaded *ELF* file and overwrites its global offset table entries with pointers to its monitoring functions. Functions themselves then mediate calls to actual library functions after they have completed monitoring if it is necessary. However, there is no direct possibility to modify LAS directly using java code, Aurasium implemented these interposition routines in C++ language, exactly as the *ld.so* loader. This approach is possible due to JNI [91] and Android's native development kit (NDK) [180] which ensures interaction between java and native C++ code.

The mediation is used to dynamically monitor the application behavior and enforce the fine-grained security policy. Aurasium introduces policies that protect the device from 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-as to escalate privilege [62] and gain the root access. All these refinements against standard built-in security policy on Android can be categorized into three main groups - *privacy policy, network policy* and *privilege escalation policy.*

The purpose of the *privacy policy* is to enhance user's privacy. This is related to accessing the private data which are available on the current mobile device, such as IMEI [125], IMSI [189], phone number, location of the device (and user probably), sms/mms messages, phone conversations or contact list, etc. The *network policy* enables finer-grained interaction with the network. For example, only particular web domains or set of IP [81] addresses can be accessed. Furthermore, Aurasium also proposes IP blacklisting provided by the Bothunter [94] network monitoring tool to harvest information about malicious devices with the specified IP addresses. The last category, *privilege escalation policy*, is used to secure the vulnerability introduced by Aurasium interposition.

Static Analysis

Aurasium's code is distributed under GNU general public license [141] and freely available on the GitHub [60] server. Currently, analysis of the code is the only way to obtain more detailed information about this framework. The most important part of the code is in the native Android application called *ApkMonitor*. It contains all the sandboxing code, that is later attached to the selected application which should be hardened. The most profound part of this application is written in C/C++ language and is called Aurasium native library [233]. It contains two types of code - the code for preparing the interposition during start-up and the code performing the mediation. The interposition code has to execute before any Android component is stated, this the solution is robust enough. Android API defines the application component for this purpose, which is called before every other component and can be used to include global initialization for an application. However, the most of the applications do not need to utilize this component, which is used in Aurasium.

Aurasium framework has to include the component called *ApiHook.java* also in the *AndroidManifest.xml* declarations. Since the implementation rewrites the LAS od the application's process, it has to be performed in the C/C++ language (*apihook.cpp*). The first operation is the analysis of the LAS and reading of the memory sections (stored in *memmap* [113] 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„-mediates functions. A "hook„ function (files with the prefix "*hook_*„ in the name) represents the second type of a native code - it performs the mediation. This function replaces the standard *libc* functions with the mediation code and delegates further processing to lower layers in the end.

Aurasium tries to minimize the amount of native code because it is generally difficult to write a test for this code. For that reason, Aurasium has all the policy logic in the Java programming language and has been built upon many helper functions in the standard Android framework. However, including Java code into existing *apk* installation package is not trivial and requires some intermediate steps. In Android, all application's Java code has to be compiled to single file called *classes.dex* which contains byte-code for virtual machine similarly to java *class* file, but contains all compiled java files. Therefore, there is a need to dis-assembly the *dex* file, insert Aurasium's sandboxing code and re-assembly it again back to create a new *classes.dex* file. Fortunately, there exists an open-source reverse-engineering tool to perform such task, which will be examined and described in the following section. This tool is also used in Aurasium framework.

Regarding *AndroidManifest.xml* file, all components started in the application have to be declared in this file and therefore, also the modification of this file is part of repackaging already discussed *apk* file. This principle of this approach is to attach application class declaration in the manifest file, which will be instantiated by run-time whenever the application is about to start. It enforces the global offset table change (delegating the Aurasium native library) 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 of the Aurasium framework are interconnected by using Bash shell interpreter [28] of Unix operating system [26]. In the first phase, the content of *apk* installation file is validated (using script *Singer.py*) and obtained using reverse-engineering tool *apktool.jar*. The next step is launching the *RegriterMain.py* the script, which injects the monitoring code. This essential script copies the *Aurasium native library* into */jni* folder and adds new declaration *<Application>* into *AndroidManifest.xml* file.

In the next part, the *apk* installation file is packed again. Finally, the last part of the system is a Python script for the application signing 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, as was discussed earlier. Thus, Aurasium re-signs applications using a new self-signed certificate maintaining a one-to-one mapping between original certificates to equivalence classes of authorship among applications. Anyhow, this case is unique, and Aurasium usually applied to standalone applications where application updates a cooperation between more applications are not common.

6.2 Design of Prototype

The design of the solution should focus on private user data, and it is restriction outside the device. This restriction should be performed without affecting the original behavior of applications, which implies dynamic policy enforcement and use of tainting mechanism. Related to the theoretical amount of work, the implementation of the prototype is limited to focus only on files and tracking it is duplicates which can also be provided to sensitive Android components which are called before leaving the system.

In the following text of this section is the design of the solution which should pretend or completely deny the leakage of the data through these sinks. The solution is designed

according to the concept of the limitation to the files only. These files are categorized into two primary groups - *public, private*. Public files are standard files which do not need any restrictions. On the other hand, the private files should be restricted, and the leakage should be impossible. The design of prototype does not include the categorization of the files. It can be done by some classification, which is not the scope of this thesis. To handle the membership files to groups, there are already exists two folders on the disk partition which are named as groups. For instance, files which are public are saved inside public folder, otherwise they are saved in private folder.

The work is built upon the Aurasium framework and accomplish the policy enforcement using the monitoring system calls. The framework contains monitoring hooks for several Android system calls which are listed in the following table 6.1.

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

Table 6.1: Aurasium intercepted system calls

Binder

In order to perform the required mediation, the part of the Android middle-ware called the Binder needs to be rewritten. Binder provides a high-level abstraction on the top of traditional, modern operating system services. Also, it also accomplishes binding functions and data from one execution environment to another. OpenBinder [87] is customized to provide inter-process communication as was described in the section 4.4. Interposition code needs to be placed in the suitable position on the original binder implementation. Therefore, there is important to understand the concepts of the mechanism and to analyze the architecture of this part of the system.

The communication between two processes is ensured by binder objects (BO), which are instances of classes that implement ioctl-based binder interface. The most important method which is defined in the interface is `transact(int code, Parcel data, Parcel reply, int flags)`. The appropriate callback method in the binder object is called `onTransact()`. The interface can be further extended by additional business operations as was described in section 4.4.

The communication is processed as follows. Each BO has a local and global identifier. The local identifier is unique in the process, and the global identifier is created when the BO is passed to another process using binder driver. Binder driver works like a network switch and persists the mapping from a local identifier to a global identifier in the table structure and translate it transparently, similarly than the mapping using ARP protocol [106]. The Binder framework communication uses the client-server model. However, the process can implement the server, as well as the client so that the communication can be still bi-directional. The binder client invokes an operation on remote binder object called binder transaction, thus, can involve sending or receiving data over the binder protocol.

In Android, the binder driver performs the communication indirectly and it is exposed through */dev/binder* file. Simple API is based on operations *open(), release(), poll(), mmap(), flush()* and *ioctl()*, etc. The first parameter is the file descriptor number which identifies currently open file and it is used in */proc/<pid>/fd/<fd>* file. The second pa-

rameter specifies the *ioctl()* command. The five important *ioctl()* commands are listed bellow.

- `BINDER_WRITE_READ` - sends zero or more binder operations, then waits (blocking waiting) to receive incoming operations and returns 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 thread will remain idle
- `BINDER_SET_REPLY_TIMEOUT` - sets the time threads will block waiting for a reply until the time out
- `BINDER_SET_MAX_THREADS` - sets the maximum amount of threads that the driver is allowed to create for that process's thread pool

The most communication is done through `ioctl(binderFD, BINDER_WRITE_READ, &bwd)` operation, where the *binderFD* is used to access the binder file and *bwd* is a structure for binder read/write operations defined in listing A.1. The illustration of the binder transaction is illustrated in figure 6.4.

The commands for a driver are called binder call (BC) and the commands for the BP are called binder return (BR) commands. Both commands abbreviation are used as a prefix of the name of binder driver commands (see the table 6.2). Each command is a couple consisting of operation code and data.

These commands (couples) are stored in the *binder_transaction_data* structure depicted in appendices listing A.2. When the transaction is inline, the data is directly stored in the structure. Otherwise, the structure contains a pointer to the data buffer. The list of available binder driver commands, which are stored in the buffers (*read_buff, write_buffer* is listed in table 6.2.

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 6.2: Binder driver commands

The binder transaction is a passing data from the client to the service, while binder reply is a passing data from the service back to the client. This behavior of binder driver interaction is in figure 6.5. The whole binder framework mechanism is transparent to the Android developer since the binder transaction is performed as a local function call using thread migration. It is ensured by the proxies and stubs, which are auto-generated helper classes from AIDL files. Proxy is the helper class performing the transformation Java code into low-level commands for the binder driver. The stub works in reverse to proxy and

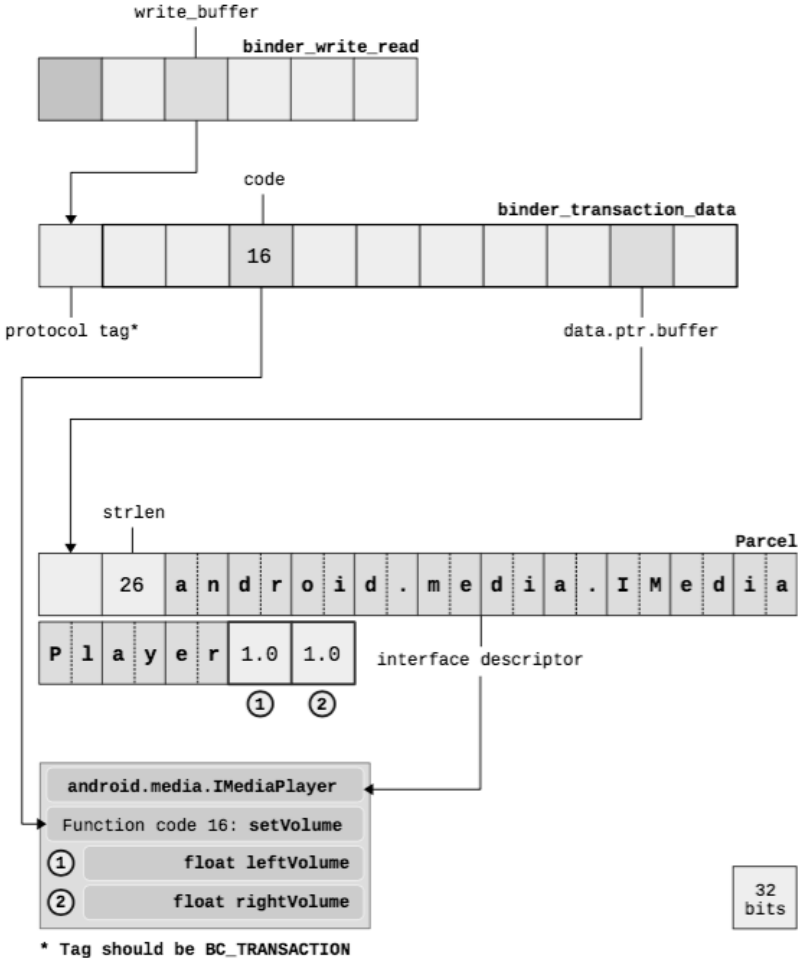

Figure 6.4: Binder Transaction [21]

automatically parses and performs read commands on the service side. The overview of this mechanism was mentioned in section 4.4 and the principle of proxy and stub is illustrated in figure 4.9.

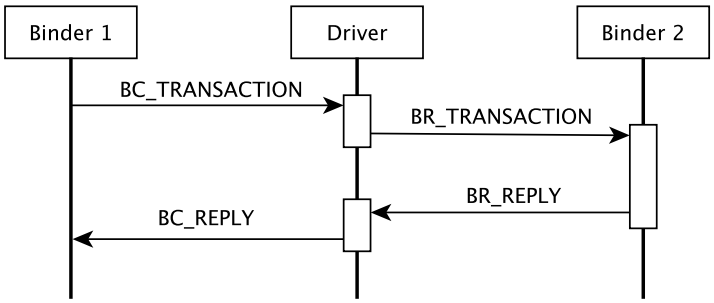

Figure 6.5: Binder Driver interaction

Since the binder driver is implemented on the low-level layer using C programming language, there is required the layer responsible for encapsulation of high-level Java objects. This is secured by *Parcel* container and corresponding *Parcelable* interface. A procedure for converting this high-level data structures into parcels is called marshaling. The mechanism of marshaling and also unmarshalling, as the reverse process of marshaling, is the responsibility of the proxy and stub components.

System Design

Design of the system is based on previously defined binder modification and Aurasium framework. Related to previous chapters it is evident that the solution is based on taint analysis principle, the tainting is based on the principle used in TaintDroid. In order to perform complete memory tainting, the tracking of each atomic memory operation is needed. This approach in programmer's point of view means each variable assignment, modification or unset have to be tracked. It is possible through monitoring of instructions on the level of machine operations. The approach introduced by TaintDroid monitors instructions on the virtual machine level., because all possibly malicious applications are run inside virtual machine environment. As was discussed, TaintDroid uses virtual taint map (VTM) which is responsible for mirroring the address space but does not contain the content of the memory. It represents the division of the memory into two groups - public and protected. The tainted files are marked in VTM before the tainting process starts. Afterwards, every copying of memory invokes copying blocks in VTM. Since the applications, which run on a separate virtual machine can exchange data, TaintDroid introduces message-level tainting principle.

This prototype is focused on the integration of two granularities of tainting the file and data level. The message-level taint principle (data exchange between components) from TaintDroid is used for the final policy enforcement. Usage of the message tainting is mainly at the final stage of the enforcement that is the restriction because Aurasium framework intercepts only single applications and cannot monitor the unhardened applications. Tainting principle on the file-level and also data-level mentioned in this section use the mentioned principle of VTM.

File-level tainting between the operating system's files and a memory can be performed in a full scope, because Aurasium framework can intercept this communication in a full scale using these system calls *fopen(), open(), write()* and *read()*. Also, the function *fopen()* is not used only for opening files, but also for obtaining the mode of the opened file. This mode is used for designing the tainting customization. When the untainted memory block is written to the tainted file in append mode, the file remains tainted. However, when the untainted memory is written to the file in write mode, the file is untainted as well, because the content of the file is overwritten by the content of the untainted memory. The operations *open()* and *read()* are use for tainting the memory blocks as well as new files. The blocks in memory filled from the tainted file are marked same as the file is marked. This means that data in memory filled from the tainted file is marked tainted as well. Moreover, data in memory are also directly propagated.

Aurasium can intercept only specific places (system calls) and not the instruction itself. It is impossible to implement full-range memory-level tainting as is introduced by the TaintDroid solution. This is replaced by the newly designed data-level tainting concept. This concept together with the file-level tainting is illustrated in figure 6.6. The content of the file is read and tagged using a hash function to assign a unique number. This tag, along with the size of a block is used during the writing unknown memory block into the file.

Each unknown memory block is tested concerning any existing hash and marked tainted if the hash matches. Subsequently, the file is marked as tainted accordingly.

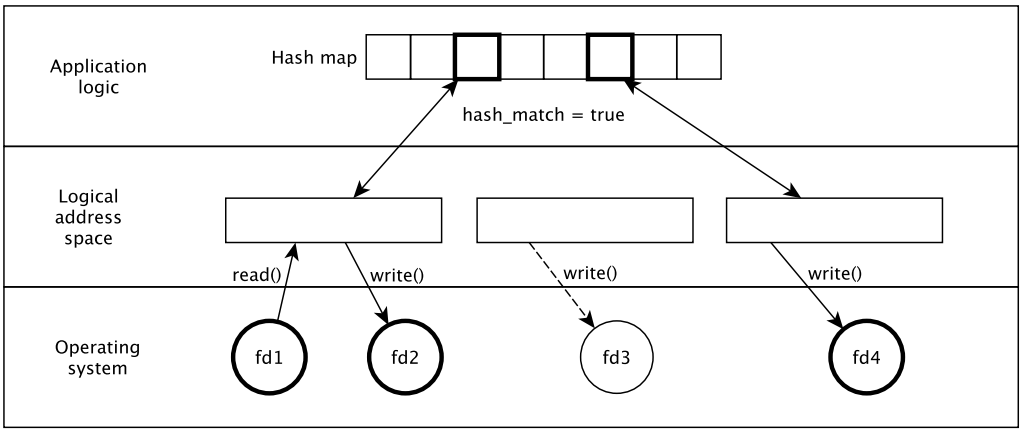

Figure 6.6: Design of data/file level tainting

In figure 6.6 are private files represented as file descriptors on the operating system level, and they have the bold border. The first file description (*fd1*) is the user selected private file that is read into the memory. In this case, the hash of the private file is computed and saved into the hashmap situated at the application logic level. When there is a command to save this content of the memory into different file represented as second file descriptor (*fd2*). This file is also marked as private and tainted. On the other hand, if the content of the memory does not have calculated hash inside hashmap it means, that this file is not considered as private and the tainting is not required. This behavior is illustrated with file description number three (*fd3*).

The final policy enforcement is performed using the interception of *ioctl()* system call. Moreover, when the `BR_TRANSACTION` command, which consists of destination component content provider, is read, all the *read()* system calls for the tainted files are in the mode of restriction. The prototype is proposed to secure the user-selected files or folders as an entity, which is intended to be invariable such as images, pictures or movie clips. Documents that are often changed can be restricted for opening, or there can be assigned individual rights for opening inside hardened applications, and the files are encrypted for other applications (unhardened ones). In this inverse mode, data is protected with unhardened applications and uncovered and possibly exploited by the hardened applications. The inverse mode is designed as optional, and it is not part of the prototype.

Unknown memory block which will be written to a 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 the file has been divided into smaller units with separate hash value. Design of data structure with the same meaning as TaintDroid's VTM has is implemented as a simple array of memory blocks. These blocks are an interconnection between the file system level and application logic performing described data-tainting. In order to implement this approach, each memory block is considered as tainted or untainted. The 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 blocks are added. Each memory block has assigned only one file which is the source of its data, one counted hash value (hashtag), but many destination files to which this data is written.

In regard to tainting customization, the file modes need to be stored, because *read()* and *write()* functions do not dispose with this information in the passed arguments. Implementation code of the designed data structures focuses on tainting process which was described in this section are expressed in listing A.3.

In regards to the configuration possibilities, the primary purpose is to allow a user to select and marked privet files and folders. Tainting based on hashing principle can be used in situations where the low memory consumption is the most crucial parameter while the content-based scanning in the case of tiles which need more robust security mechanism. In order to achieve better decision, the selection mechanism was prepared in which the user can choose the private file/folder.

In some specific cases, there is also need to disable the tainting at all due to performance slowdown or extreme battery power consumption. The restriction can be performed explicitly as well as dynamically. For the protection against theft or unauthorized users the specific restriction can be used, sometimes it is called static restriction. For instance, this situation can be described as parental control. Dynamic enforcement can be realized using confirmation screen, and it is useful for its flexibility. It is also appropriate to consider the configuration of permanent restriction where the selected protected files cannot be even read by the unhardened application which using encryption of the files and only hardened applications have the key for decryption. The configuration settings can be assigned to all applications centrally or individual applications separately. Graphical screen with the configuration settings can be injected into hardened applications using the first phase of Aurasium framework - modifying the installation package. In this case, the application configuration is individual for each application. In order to have the central control over all hardened application, the graphical screen is resolved as a single central application, which needs to be installed before any modified application. This approach is preferred according to proof of concept solution and also the ability to control multiple applications (modified applications only) at one place.

6.3 Implementation

In order to implement the prototype solution the development environment needs to be set up, the only working combination for the Aurasium project in these days is Windows operating system, Linux operating system and Eclipse integrated development environment (IDE) [49]. Moreover, the Android studio plugin [66] is required to install into Eclipse IDE, which is provided by the Android maintainers. The reason for the restriction of using Eclipse IDE is the full support of Android NDK [180] and Aurasium's dependency on the Android package directory structure used by older Ant building system [103]. Some parts of the Aurasium system are written for Windows Cygwin Linux environment [179]. For instance, the Makefile [208] for creating code bundles for repackaging. Windows platform is also used for the Eclipse IDE, virtualization of the Android operating system and the whole development. The Linux is primarily used for standalone repackaging scripts which are the part of the Aurasium and are strictly connected to this operating system.

The next stage is to support at least Android version 4, which is considered as revolutionary compared to older ones. The next versions of Android are built on the changes in this version, and the following updates will not be so complicated. The currently supported version of the Aurasium framework is 2.3.3. In this case, the changes need much work, and this is the challenge of this work. The issues related to this improvements are application crashes during run-time, wrong initialization and linking of the JNI library and or more

complex issues related to dynamic linking in the Android 4 and the following Android 5 version. The result of this stage is the functional built-in *ApkMonitor* application running on the Android version 4, 5, and 6, which are the most wide-spread Android versions in these days. A particular part of the second phase is the reconstruction of the repackaging script, which generates the corrupted installation packages as default. Even the process finishes successfully. The package is corrupted.

The last part of the implementation covers the selection of examined several open-source, an available application which is simple enough to enable editing and sharing data. The most appropriate turned out to be the simple file manager *OI File Manager* [169] (application 1), which is publicly available on the Google Play [145]. This application has been manually modified. In more details, the *ApkMonitorActivity.java* has been injected instead of the original version, this java file is called after the mediation of system call is started. The information base for this step has been introduced in section 6.1.

Aurasium is capable of intercepting various system calls. Only limited set of them is relevant to the topic of this thesis. Those are the system calls *ioctl(), open(), fopen(), read()* and *write()*, and they were introduced in the section 6.2. The most important collected information is the time when the Android API performs the calls and the content of the passed arguments. In order to implement the required behavior, there is the need to follow some parameters of the system calls. These parameters are listed with the functions in the table 6.3 of function prototypes.

Function prototypes
`int open(const char *pathname, int flags, ...);`
`FILE *fopen(const char *path, const char *mode);`
`ssize_t read(int fd, void *buf, size_t count);`
`ssize_t write(int fd, const void *buf, size_t count);`
`int ioctl(int fd, unsigned long request, ...);`

Table 6.3: Function prototypes of system calls

System calls have been examined using *gdb* debugger, *LogCat* messages and logging to a file. Debugging is the best method for tracking the sequence of the code in the time, *LogCat* messages are the most usable, but the logging to a file is the most appropriate way because there can be persistently stored also a very long sequence of the calls for the further analysis.

Function *ioctl()*

However the Android project repackages the application singly, there is no possibility to track the full communication from the one binder object to another and vice versa as was illustrated shown in figure 6.5. The communication can be tracked between hardened application and the binder driver as is shown in figure 6.7

Even the implementation of the Aurasium mediation is performed via rewriting the *ioctl()* function and its operations as the result on the call commands `BC_TRANSACTION` and `BC_REPLY`, induced transactions from the hardened application are intercepted only as the remote calls `BC_TRANSACTION` and `BR_REPLY`. The another half of communication is ensured within the delegated original *ioctl()* function which is outside the application as is depicted in figure 6.8.

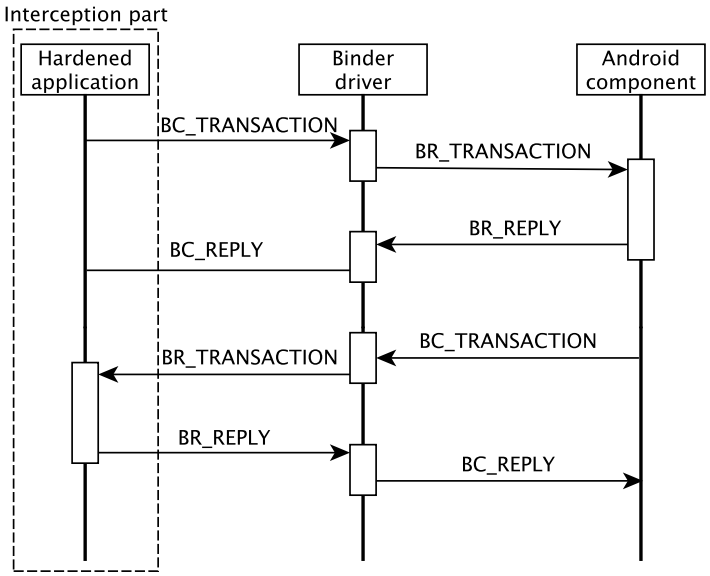

Figure 6.7: Interception overview

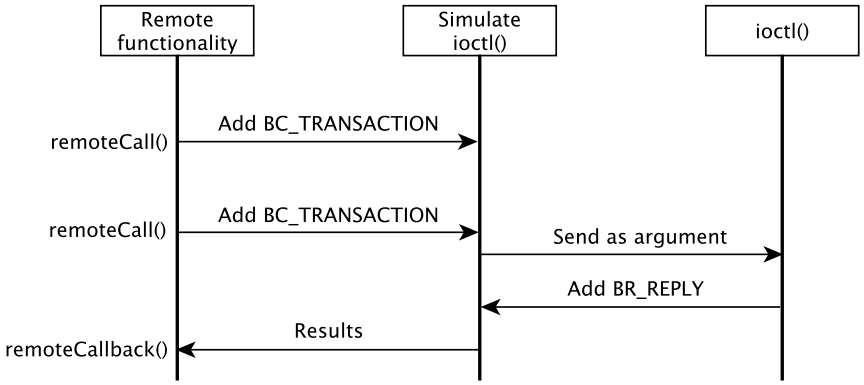

Figure 6.8: Interception overview - detail view

During the attempt of application to use standard Android API, such as share data through available channels like internet, there is sent an intent with action *ACTION_SEND* or *ACTION_SENDTO* to reference monitor. The application does not perform this operation itself, even if it has declared required permissions in its manifest file. There are capture messages which are sent to various external components or received in the hardened application. The majority of messages is in the outward direction and are also invoked during the idle state. The list of the most important ones are listed in the table of captured call transactions 6.4 and table of captured return transactions 6.5.

However, some internal components are called by external components. It is the case of the following components, which are externally triggered by the Android system or external application. The external component *IContentProvider* sends the messages to perform specific user operation such as querying data provided by the hardened application. Therefore these messages are suitable triggers for security policy enforcement.

Interface	Description
IPackageManager	Class for retrieving various information related to the application and packages that are installed on the current device.
IActivityManager	Interact with the overall activities running in the system.
IWindowManager	Interface for communication between application and the window manager.
IServiceManager	Interface grants basic operating system services, message passing, and inter-process communication.
IInputManager	Interface provides information about input devices and key layout.
IPowerManager	Class ensures a control of the power state of the device.
IWindowSession	Interface that convey the communication between application and the window manager.

Table 6.4: Captured call transactions

Interface	Description
IInputContext	Interface from an input method to the application, allowing it to provides modifications on the current input field and other interactions with the application.
IContentProvider	It encapsulates data and provides it to applications. This is used only if the application needs to share data.

Table 6.5: Captured return transactions

Functions *open()* and *fopen()*

The difference between functions *open()* and *fopen()* is that the first one is a system call function, while *fopen()* is a high-level wrapper which uses buffering and simple interface - it is usually represented as a library call in C language. During implementation and experiments, there was captured calling of the function *open()* and only after the first start of the application. Since opening and closing files are time-consuming operations, Android manages to keep the files open over the entire life-cycle of the application and even after the opening another files. Opened files are closed when the application is shut down, which does not mean the idle state of the application, but the shut down of the process end removal from the memory.

Functions *read()* and *write()*

The system provides the ability to read the content of the file into the logical address space of the process and also reverse operation that is writing the content of the memory into the file. Since these functions are low-level, the whole file is read or written at once when one of this operation is called. In the case of application 1, the whole content of the files in the opened folder is read before the real opening of any file from the folder.

Logging

All previously defined functions have been monitored and captured through the logging mechanism, specifically the logging to the file. As was discussed before logging to the file is the most appropriate mechanism to capture the whole sequence of system calls. An-

other possible approach can be the Android LogCat messages or debugger. According to the amount of data and the need of further analysis the logging to the file is the only method which covers all required parameters. In regards to the logging mechanism, the one file of system calls is not considered as well-arranged and for this purpose insufficient. In order to define the clean structure of the log file, the proposed log files define the division into following categories (and also into different files) listed in table 6.6.

Log file name	**Description**
log.txt	Monitoring of system calls and related memory blocks.
log_taint_map.txt	Monitoring of snapshots of taint hashmap.
log_java.txt	Monitoring of Parcel content for final restriction.
log_error.txt	Additional logs with the error priority.
log_debug.txt	Additional logs with the debug priority.

Table 6.6: Summary of log files

Tainting Principle Used in Prototype

Related to the Aurasium framework which is divided into few parts, as was described in this chapter, the resultant solution of the prototype is follow the same principles. Therefore, the resulting solution consists of implementation in native code, byte-code, and scripting languages. The implementation in native code provides fast handling of tainting mechanism. Byte-code is primarily used to access the *ioctl()* function for required restriction and also for the implementation of configuration, see section 6.4. Scripting languages such as Python or Bash are used for repackaging phase, injecting the code, to sing the application and to create an installation package.

About tainting principle, the solution is implemented in C/C++ programming languages in order to achieve the best efficiency. The life cycle of the hardened application starts with the creating of log files or overwriting the existing ones when the application is repeatedly started. The next step is to load the configuration (see section 6.4), after that, the application is in monitoring mode. The main purpose of the monitoring is to track appropriate system calls such as *open(), read(), write()* and *close()*, which leads to changes in the content of the proposed tainting structures. This principle is illustrated in figure 6.9.

The resulting code is compiled into the single module into resulting *apihook.o* object file. The connection is then ensured by using the delegation mechanism and calling the corresponding handling functions before and after the current system call operation. The code of Aurasium framework can be rewritten to another version and reconnected easily. The application has to track its state statically, which implies to the usage of static variables. Unfortunately, even the C/C++ compiler retains the variable store during the whole application run, the scope of the variable use is limited according to initialization point. When the variable is initialized globally, it is not available and or accessible outside the file. Therefore, there is necessary to use global variables, which are included in all required files inside header file using keyword *extern*.

In order to create structures for tainting the C++ standard library is used, specifically the *std::vector* is used for the list of user-selected private files, the list of tainted files, the list of memory blocks and the lost of blocks of stored information called small blocks. The memory block represents a specific part of logical address spaced of a process whose content has been obtained from on specific file - source file. A memory block is defined with

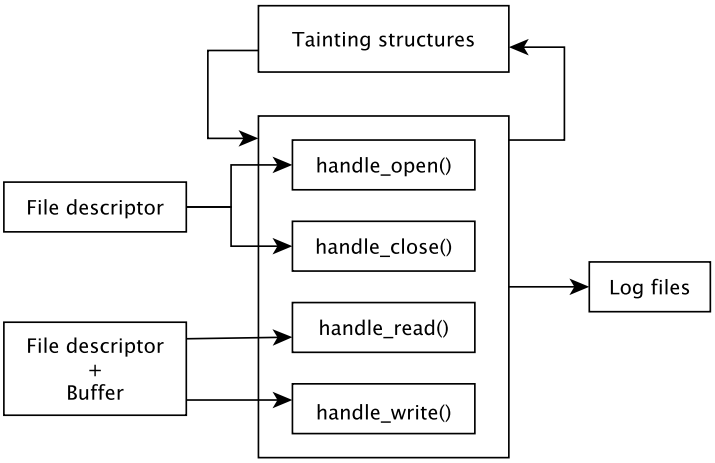

Figure 6.9: Overview of the proposed tainting system

a start address and its size and is linked to one counted hash value, several small blocks and source files. Another data structure uses to store the information about open files is *std::map*. The key values of the structure are the file descriptor numbers of the currently open files. Its usage is primarily for efficiency, because the same information can be obtained from the */proc/<pid>/fd/<fd>* file. The map is used in *read()* respectively *write()* function, because the user selects the file paths while this system calls use the file descriptor.

Tainting principle starts with an empty list of tainted memory blocks and captures the *open()* system call. When this event occurs, the necessary information about a file is stored in a map of open files. If this opened file is the user-selected private file, the *read()* event causes the creation and storing the new tainted memory blocks into the list. There is stored 256 bites long hash calculated by SHA-256 [158] hashing algorithm for the file-based scanning. The other approach - content-based scanning is implemented using small blocks, as was described earlier. The small block represents the part of the file content that is used for memory tainting. When the stored small block is found in the logical address space during the writing the content of the memory to the file then the resulting file is marked as tainted as well. The size of the small block is defined during the compilation process and depends on the usage of the application. The size should be sufficiently large to satisfy the probability of veracity of the statement, see definition 6.3.1. The size should be as large as possible, but small enough in order to provide sufficient granularity for tainting the parts of files. It is influenced by the variability of the media types. For instance, the images used to be more variable than plain text files.

Definition 6.3.1. Let set $F = \{F_1, F_2, \ldots, F_n\}$ is a set of private and tainted files, where n is the amount of theses files and $\forall x \in \{1, 2, \ldots, bAmount(F_d)\} : b_x(F_d)$ is a Small Block of a file F_d, where $bAmount(F_d)$ is an amount of created Small Blocks for the file $F_d, d \in \{1, 2, \ldots, n\}$ and b_x is a function which maps a file to its x-th small block.

Finally, let be a reflexive, symmetric, and transitive relation R that is a transitive closure of a relation which contains all couples of two files that are dependant in a way one has been created from another. Then, $\forall d_1, d_2 \in \{1, 2, \ldots, n\} \forall x \in \{1, 2, \ldots, bAmount(F_{d1})\} \forall y \in \{1, 2, \ldots, bAmount(F_{d2})\} : [(b_x(F_{d1})) = b_y(F_{d2})] \Rightarrow (F_{d1}, F_{d2}) \in R$.

In other words, the statement definition 6.3.1 claims, that if there is found a match between two small blocks, it should indicate that corresponding files are the same or one file has been created from another. Unfortunately, there can still be two files which have the similar or even the same content and the creation of these files was done independently.

The mode of the open file becomes an essential part of the process of tainting files. It is related to the particular case when the *write()* function is called to write the content of the memory which is not tainted. When this memory is not tainted, and the content is written to the private file, which is opened in append mode, the file remained tainted and marked as private. The reason is that the previous content of the file is considered as private even that the content of the memory that is appended to this file is not considered as private. The different situation occurs, when the private file is opened in the standard write mode. In the same situation (untainted content of the memory is written to the file), the file is not marked as private, because the classical write mode will rewrite the content of the file with the new content of the memory, which is considered as not private (not tainted) and in this case the resulting file is not marked as tainted anymore.

The main operation is the estimation of the presence of private data. Using file-tainting, the hash of the block, which is being written, is counted and compared to all stored hashes in the hashmap structure. In the case of content-based scanning, all small blocks of all memory blocks are compared on every position of the current block. This principle is depicted in figure 6.10.

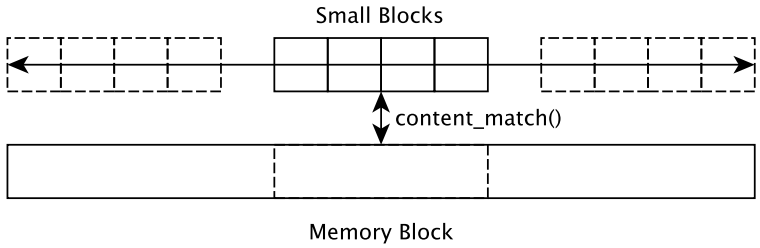

Figure 6.10: Principle of content-based scanning

The restriction of the tainted files is interposed in two places. The first one is the hardened application and the application which performs the sharing action. The second one is between the external application and the file itself. These types of restriction are referred as a restriction methods in the configuration activity. The first type of this restriction method is called *communication mediation* and is based on the Java programming language and it implements the prepared Aurasium's interface and parsing mechanism. This code is therefore interlaced with the original one in the *APIHook.java* file. Java code operations require the knowledge of which files are considered as tainted. In order to fulfill this requirement the methods *IsTaintedFile(), GetRestrictionMethod()* and *GetRestrictionType()* have been implemented using JNI framework. This implementation also includes the global variables definition in C/C++ code.

For instance, the list of variables defined for this purpose contains `SCANNING_TYPE`, `RESTRICTION_TYPE`, `RESTRICTION_METHOD`, `THREAT_LEVEL`, and `ENABLE_LOGGING`. The restriction is invoked during the Aurasium's `on_BC_TRANSACTION` callback function. The life cycle of this approach starts with the checking of a descriptor. If the name of the descriptor is equal to *android.content.IContentProvider* (described in the table 6.5) and the transac-

tion name is *QUERY_TRANSACTION*, then the hardened application's *query()* function is called. Illustration of this process is described in figure 6.11.

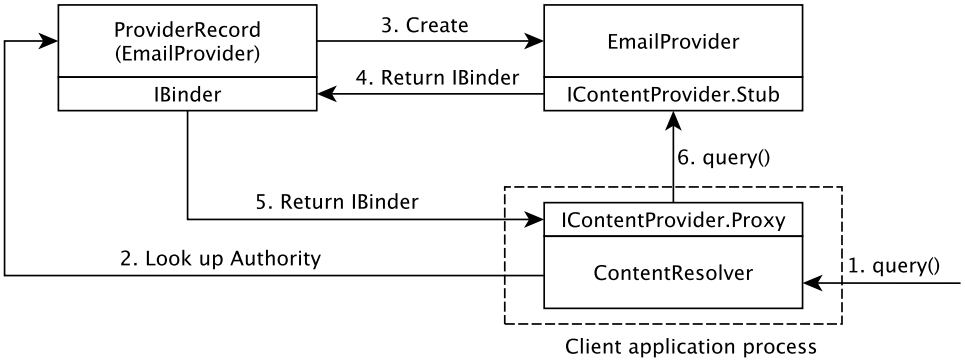

Figure 6.11: Content provider interaction

The second restriction method called *File protection* provides the restriction outside of the hardened application. It is related to invocations of the application by hardened applications and specifically by sending Android's `SEND_ACTION` command. The implementation of an external application which responses to such action command depends on its author. It can access the shared content using standard *ContentProvider* API, or it can access the files directly on the file memory system. For instance, this is the case of applications such as *Google+, Hangouts, MailDroid*, because the previous restriction is not efficient. The solution is the extension of the original application design with the falsified files. When the application wants to share the file, this file is moved to another location, and the new empty file is created in the same directory which is available to access by other applications instead of accessing the original file. The name of a backup file is usually created by adding the dot prefix and *pe* suffix. For instance the example of backup file with the name *file* is */pat/to/file/.file.pe*.

Another approach can be creating the backup file in the specially protected location. However, the lifetime of this file should be as smallest as possible, especially during the operation of external sharing application. In this prototype, the protected file is returned to its original state immediately after the hardened application is restored and there is new *open()* system function call on the file. In the case of chosen Application 1, the process is transparent, and the file is not visible to the user. The steps of implementation usually follow this order. Arguments from the *query()* function are read as the first step. After that, the content of the second argument - unified resource identifier is compared to the all user-selected tainted files. The restriction is then performed according to user settings defined in the configuration or by the configuration application. The communication can be restricted implicitly, or the configuration dialog is shown.

In order to ensure the restriction mechanism, the new parcel is created, and this parcel replaces passed arguments with the blank values. Regarding to the designed explicit restriction, this is implemented as the reaction on the *read()* function event. Moreover, the read buffer is overwritten or cleaned. This approach can be achieved by a modification of the operation on the *buffer* parameter of this system call. In the case of overwriting there is modified only the content referenced by the third parameter.

6.4 Configuration

Configuration is implemented as standalone Android application that is capable of defining the protected private files and administration of configuration. The selection of private files is provided by open source project *Android file dialog*. The graphical user interface is divided into two main basic parts called fragments. The first fragment informs the user about selected private files, and there is also a possibility to change the selection. The second fragment defines the administration of configuration for the hardened applications. The principle of communication with the hardened applications is based on configuration file usually called *private_files.conf* and the content, as the name prompts, contains the list of selected private files. This file is currently located inside folder */data/data/cz.vutbr.aron.privatefiles/files*, as it is the default location used by function *onFileOutput()* that is used for opening the private files associated with the main context of the application package. The configuration file is opened within `MODE_WORLD_READABLE` mode according to access by the configuration application and also by hardened applications.

This method of manipulating with data is convenient for prototype or testing purposes only. It usually works on the level of operating system and can be therefore directly accessed by interposed hook functions written in native code. The hardened applications load and maintain the state of current configuration at the beginning of the process and during its run-time. Manipulation with the configuration file is insured by *Utility.java* file provides the basic functionality such as *loadFromFile()* and *saveToFile()*.

The fragment with the configuration setting is extending the *PreferenceFragment*. This implementation of Android API automatically creates the graphical interface known from the Android standard setting screens and remembers the settings utilizing the Android *PreferenceManager*. Moreover, there is also the implementation of the feature that listens to changed-preference event and maps the saved, shared preferences immediately into the configuration file. Configuration screen can set the tainting, type of restriction, a method of restriction, type of data falsifying and logging feature. In order to achieve this ability, the configuration file has the first line definition of mentioned parameters. The structure of the file is depicted in listing 6.1, in which the shortcut *Restr* means restriction.

```
<Tainting><Restr. Type><Restr. Method><Threat Level><Logging>
  <Absolute Path to the Private File or Folder>
  <Absolute Path to the Private File or Folder>
...
```

Listing 6.1: The structure of configuration file

The protection of configuration application and also configuration files are not part of this prototype, and in this case, it can be the weak point of the solution. However, this prototype should determine that the proposed concept can implement and the behavior of the permission enforcement is dynamically changed during the file manipulation.

6.5 Implementation Limitation

This section describes the limitation of proposed implementation. The implementation does not cover the protection against leakage in all circumstances. This section describes a few

cases in which the protection can be weak or does not work at all. Note that this thesis and mainly this chapter is focused on implementation of the prototype, which should define proof of concept. The implementation can contain bugs, errors or vulnerabilities. Let's discuss three main topics related to limitation of the implementation.

Modification of Applications

In regards to concept, the implementation is defined as a layer between an operating system and the user. Moreover, the installation and usage of the prototype do not require administrator access (root access). Owner of the device (or the user) can choose which applications communicates with the underlying operating system via the prototype. The BYOD principle is defined as using a personal mobile device in the working environment. There should be a definition of control which application is required to be modified by the Aurasium framework with the prototype implementation. In order to circumvent any required restriction, the application can be omitted from that modification or removed from the device and installed again without related system function hooks.

Pre-installed Applications

The second limitation is related to pre-installed applications on the system level. Vendor of the operating system provides a set of applications that are present on the platform. Applications usually persist on the platform and they cannot be removed by standard principle. These applications can be modified with the script and system calls hooks can be added, but this needs to be done on the image of the related operating system for the specific mobile device. Also, this image needs to be installed on the mobile device. Related to BYOD, the user usually has already functional mobile device with an already installed operating system.

Tainting of File Content

Implementation of tainting the file content is handled on the prototype level. The content of the file is split into small blocks, as was already discussed earlier in this chapter. These small blocks have a specific size, and for each block, the SHA-256 hash is calculated. The hash value is saved and compared during the write operation. The result of the comparison is the category of the file - public or private. In order to circumvent this protection, the smaller amount of unit needs to be shared/send. For instance, the text document can be shared through any channel split by letters. This technique takes as long as the document has letters.

6.6 Model of Implementation

The formal definition of implementation can be described in the very similar deterministic FSM as was required. The formal model expresses the file taint mechanism described in the previous sections of this chapter. As was already described in the concept of this work the files should be divide into two categories - public and private. The decision logic is not part of this work, but in the prototype, the user has the power to select files and mark them as private. For the model specification, this action is considered as automatic within the opening of the file, and the file remains in the same category for the whole life-cycle of the automaton.

The FSM defines the behavior of one specific file on the mobile device. In order to handle all available files (and new ones as well) the amount of automaton is equal to some files on the mobile device. Therefore the same principle as in the model of required behavior is used - the FSM is simulated by the TM with two tapes and two independent reading heads. The first tape (input tape) contains the input file operations (with the same transformation as was described in definition 5.3.1), the second tape (state tape) consisting of FSM state on the specific file. Note that the formal model is not the precise model of implementation. The reason is that the model is used in the verification process and the results of verification need to be computed in a reasonable amount of time.

The formal definition of finite state automaton for implementation solution is defined as $FSM = (Q, \Sigma, \delta, s, F)$, where

- $Q = \{s, Spu, Spr, Wpu, Wpr, Cpu, Cpr\}$ - is a finite set of states
- $\Sigma = \{open\_public, open\_private, read\_public, read\_private, write\_public, write\_private, share\_content, copy, seek\_position, close\_public, close\_private\}$ - is a finite input alphabet
- δ - is a state-transition function of type $Q \times \Sigma \to 2^Q$
- $s \in Q$ - is an initial state
- $F \subseteq Q, F = \{s, Cpu, Cpr\}$ - is the set of final states

Figure 6.12 describes the finite state automaton of the implementation. The top half is defined as working with public files, and the bottom part is for working with private files. Note that the implementation handle this operation according to their parameters or file path, but for the formal model is more transparent with two possible transitions. As was described the decision logic in the state s is defined by the user selection. The history of the file persists during the first transition from state s into one of the possible states - Spu, Spr. The meaning of the states is: s - start, Spu - start public, Spr - start private, Wpu - work with public, Wpr - work with private, Cpu - closed public and Cpr - closed private.

The main logic is provided during operations (transitions) on states Wpu or Wpr, which defines the implementation logic layer, already presented. This layer consists of guarding the content of opened files, working with files memory blocks and managing file operations.

Transitions or state-transition function defines the user operations with any application. The sequence of user operations can be described as a sequence of system functions which are already defined as the set of input alphabet Σ. The definition of state-transition function δ is depicted in table 6.7.

Note that the transition called *share* is one of all possibilities for sharing provided by the operating system and also installed applications. According to many possible sharing methods, the model has only one transition name which wraps all possible choices. Moreover, this sharing method is available during public file operations, because all sharing methods use *ioctl()* function, which can be stopped inside hardened application by the implementation of the prototype.

This FSM defines the behavior for one specific file on the mobile device. To control all possible files, the TM can be used, and the same principle as during the model of required behavior is defined here as well. Therefore the model of the TM is the same, and it is illustrated by figure 6.13. The whole formal definition of Turing machine as a model is $TM = (Q, \Sigma, \Delta, \Gamma, \delta, s, F)$, where

$\delta(s, open\_public) = \{Spu\}$	$\delta(s, open\_private) = \{Spr\}$
$\delta(Spu, read\_public) = \{Wpu\}$	$\delta(Spr, read\_private) = \{Wpr\}$
$\delta(Spu, write\_public) = \{Wpu\}$	$\delta(Spr, write\_private) = \{Wpr\}$
$\delta(Spu, close\_public) = \{Cpu\}$	$\delta(Spr, close\_private) = \{Cpr\}$
$\delta(Wpu, read\_public) = \{Wpu\}$	$\delta(Wpr, read\_private) = \{Wpr\}$
$\delta(Wpu, write\_public) = \{Wpu\}$	$\delta(Wpr, write\_private) = \{Wpr\}$
$\delta(Wpu, write\_private) = \{Wpu\}$	$\delta(Wpr, seek\_position) = \{Wpr\}$
$\delta(Wpu, close\_public) = \{Cpu\}$	$\delta(Wpr, close\_private) = \{Cpr\}$
$\delta(Cpu, open\_public) = \{Spu\}$	$\delta(Cpr, open\_private) = \{Spr\}$
$\delta(Wpu, share\_content) = \{Wpu\}$	$\delta(Wpu, seek\_position) = \{Wpu\}$
$\delta(Wpu, copy) = \{Wpu\}$	

Table 6.7: Definition of state-transition function (δ).

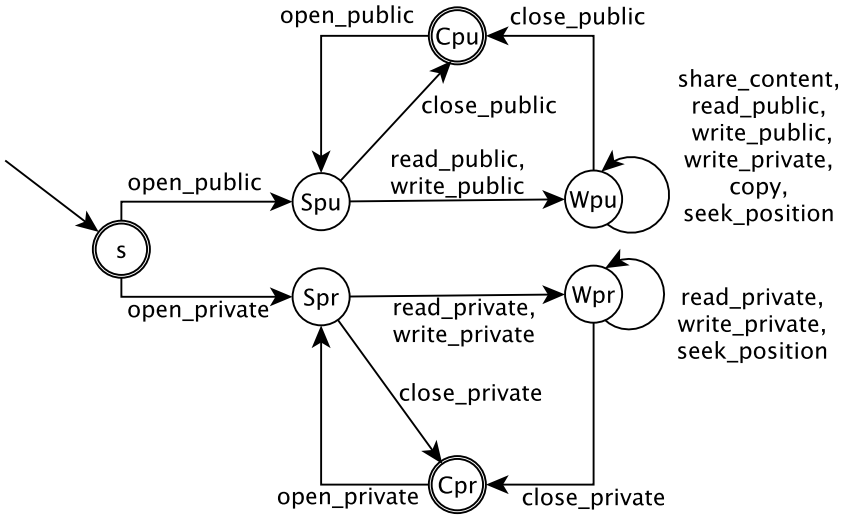

Figure 6.12: Finite state machine defines the implementation behavior for file operations.

- $Q = \{1, 2, 3, 4, A, R\}$ - is a finite set of states.
- Δ - is a blank symbol of the tape denoting the unused space on the input tape.
- $\Sigma \setminus \{\Delta\}$ - is the set of input symbols, that is, the set of symbols allowed to appear in the initial tape contents. This alphabet appears on the first tape only.
- $\Gamma = \{Spu, Spr, Wpu, Wpr, Cpu, Cpr\}$ - is a finite set of tape alphabet symbols which appear on the second tape only.
- $\delta : (Q \setminus F) \times \Sigma \bigcup \{*\} \times \Gamma \bigcup \{*\} \rightarrow Q \times \Gamma \bigcup \{L, R, \_\}$ - is a transition function, where $*$ is any symbol, L is left shift, R is right shift, and _ is no-operation symbol.
- $s \in Q, s = 1$ - is the initial state.
- $F \subseteq Q, F = \{A, R\}$ - is the set of final states.

The allowed input symbols on input tape are the files represented in the form of sequences of symbol I, as was already defined in definition 5.3.1, and the name of operations on these files. Legal input can be defined as regular expression 6.1. After reading the symbol from the input tape (first tape of TM), the reading head moves to the right automatically when the reading possible according to state-transition function.

$$\Sigma = I^{+}[open\_public|open\_private|read\_public|read\_private|write\_public \\ |write\_private|share\_content|seek\_position|copy|close\_public|close\_private] \tag{6.1}$$

The second tape of the TM simulates finite state automaton for implementation in each cell of this tape. Therefore the cell of the second tape consists of empty symbol Δ denoting that with the specific file representing the sequence of symbol I and pointing to this cell with empty symbol was not already used (opened). Otherwise, the symbol of the same cell can have only one of the allowed symbols defined by Γ.

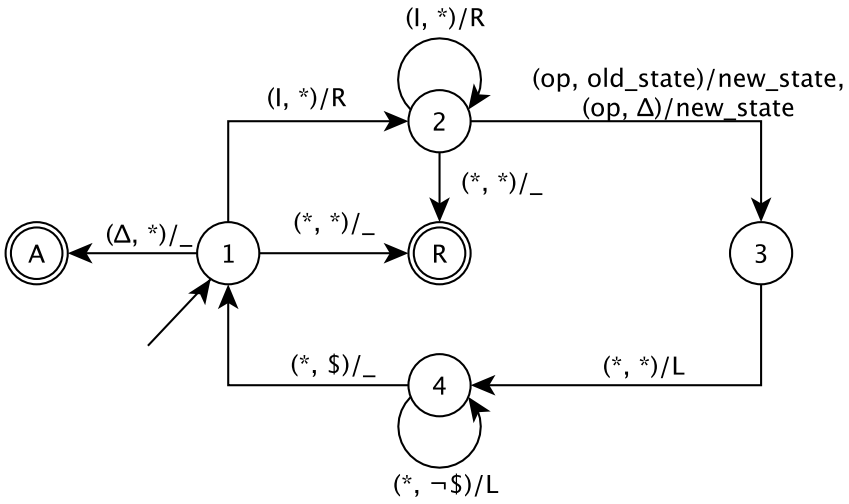

Figure 6.13: Turing machine defines implementation behavior

Transition function between states of TM is defined by the δ in the table 6.8. Moreover the symbols (_) and (∗) have the specific meaning. The symbol of star denotes the wildcard for any symbol on the tape. On the other hand, the symbol (_) means no-operation on the reading head, in other words, the reading head remains in the same position.

Turing machine for implementation behavior halts in two possible use cases. The first one defined by the reading the whole input tape and when there is no input (the symbol Δ is on the input tape) thus the transition into the accepting state A is performed.

The second halting the TM is during the wrong input tape symbol, such as the operation without the prefix of symbols I or not allowed the sequence of steps defined in the finite state automaton for implementation. In this case, the transition into rejecting state R is performed.

$\delta(1, I, *) = \{(2, R)\}$	$\delta(1, \Delta, *) = \{(A, \_)\}$
$\delta(1, *, *) = \{(R, \_)\}$	$\delta(2, I, *) = \{(2, R)\}$
$\delta(3, *, *) = \{(4, L)\}$	$\delta(4, *, \neg\$) = \{(4, L)\}$
$\delta(2, open\_public, \Delta) = \{(3, Spu)\}$	$\delta(2, open\_private, \Delta) = \{(3, Spr)\}$
$\delta(2, open\_public, Cpu) = \{(3, Spu)\}$	$\delta(2, open\_private, Cpr) = \{(3, Spr)\}$
$\delta(2, read\_public, Spu) = \{(3, Wpu)\}$	$\delta(2, read\_private, Spr) = \{(3, Wpr)\}$
$\delta(2, write\_public, Spu) = \{(3, Wpu)\}$	$\delta(2, write\_private, Spr) = \{(3, Wpr)\}$
$\delta(2, read\_public, Wpu) = \{(3, Wpu)\}$	$\delta(2, read\_private, Wpr) = \{(3, Wpr)\}$
$\delta(2, write\_public, Wpu) = \{(3, Wpu)\}$	$\delta(2, write\_private, Wpr) = \{(3, Wpr)\}$
$\delta(2, write\_private, Wpu) = \{(3, Wpu)\}$	$\delta(2, copy, Wpu) = \{(3, Wpu)\}$
$\delta(2, close\_public, Wpu) = \{(3, Cpu)\}$	$\delta(2, close\_private, Wpr) = \{(3, Cpr)\}$
$\delta(2, close\_public, Spu) = \{(3, Cpu)\}$	$\delta(2, close\_private, Spr) = \{(3, Cpr)\}$
$\delta(2, seek\_position, Wpu) = \{(3, Wpu)\}$	$\delta(2, seek\_position, Wpr) = \{(3, Wpr)\}$
$\delta(2, share\_content, Wpu) = \{(3, Wpu)\}$	$\delta(4, *, \$) = \{(1, \_)\}$

Table 6.8: definition of state-transition function (δ).

6.7 Summary

This chapter presented the system design that should satisfy the model of required behavior. The description covers the implementation details about the prototype of the solution, which defines its behavior that is required. At the beginning of the chapter, the framework which was used for the prototype was introduced with its capabilities and limits. The principles of the proposed solution for the specific mobile platform was defined in technical aspects and programming point of view. Design and the implementation of the prototype define the required behavior and discuss new approaches to the solution. There were presented two types of tainting principle, and one of them was implemented.

The second part of this chapter describes the formal model of the implementation. In order to simplify the verification process, the model of implementation was defined in the same format like the model of required behavior has. The Turing machine defines the ability to model the unlimited amount of files, and the behavior is described by the finite state automaton, that is simulated by the Turing machine.

Next chapter discusses the verification process, that should confirm or deny the satisfaction of implementation model with the model of required behavior.

Chapter 7

Formal Verification

In the context of software systems, formal verification [156] is the act of proving or disproving the correctness of proposed systems or underlying parts of the intended systems to a particular formal specification or property using formal methods (particular kind of mathematically based techniques for the specification).

The verification of software systems is done by providing a formal proof of an abstract formal model of the system, the correspondence between the formal model and the nature of the system being otherwise known by construction.

Related to the thesis, the verification process was chosen in order to prove or disprove that the implementation solution satisfies the required behavior. There exist other approaches of proving behavior between two models in a less formal way, such as debugging or testing for required behavior and demonstration of its results with few examples. Verification seems better format for that proves related to the results of existing tools. These tools usually provide the answer about the required properties satisfaction. Since these properties are usually satisfied one sentence is enough. Otherwise, the counterexample is usually provided, which is more helpful than just one sentence about disproving, which is the result of other techniques (debugging or testing). Next sections discuss the verification approaches and selection of the conventional method and software for verifying implementation of the prototype (its model) and the model of required behavior. Description of existing software tools dedicated to verification is also part of this chapter. Therefore, there is no need to write own verification tool, because there are lots of existing solutions aimed at this area.

7.1 Verification Approaches

The verification process can be performed in various formats related to the formal model definition. One approach and formation is model checking [57], which consists of a systematically exhaustive exploration of the mathematical model. Mathematical models used for model checking are possible finite state machines, but also for some infinite models where infinite sets of states can be represented efficiently finitely by using abstraction or taking advantage of symmetry. It consists of exploring all states and transitions in the model, by using smart and domain-specific abstraction techniques to consider whole groups of states in a single operation and reduce computing time.

The properties to be verified are often described in temporal logic [84], such as linear temporal logic (LTL) [203], property specification language [136], or computational tree

logic (CTL) [97]. The advantage of model checking is that it is often fully automatic. However, its primary disadvantage is that it does not work in general scale to large systems, symbolic models are typically limited to a few hundred bits of state, while explicit state enumeration requires the state space being explored to be relatively small.

There exist many tools for model checking verification in these days. Besides these tools, general purpose formal verification tool can be used to verify two models. However, it is better and more intuitive to use one of the existing tool explicitly designed for verification two models. One of the well known general purpose verification tools can be considered Spin [108]. Spin targets the efficient verification of multi-threaded software, not the verification of hardware circuits. The tool supports a high-level language to specify systems descriptions called PROMELA (short for PROcess MEta LAnguage). Spin has been used to trace logical design errors in distributed systems design, such as operating systems, data communications protocols, switching systems, concurrent algorithms, railway signaling protocols, control software for spacecraft or nuclear power plants. The tool checks the logical consistency of a specification and reports on deadlocks, race conditions, different types of incompleteness, and unwarranted assumptions about the relative speeds of processes.

Another tool for modeling, validation, and verification of real-time systems called Uppaal [36] is a parallel composition of timed automata extended with data types (bounded integers or arrays). It is appropriate for systems that can be modeled as a collection of non-deterministic processes with finite control structure and real-valued clocks, communicating through channels or shared variables. Typical application areas include real-time controllers and communication protocols, in particular, those where timing aspects are critical. Uppaal consists of three main parts: a description language, a simulator, and a model-checker. The description language is a non-deterministic guarded command language with data types (e.g., bounded integers, and arrays). It serves as a modeling or design language to describe system behavior as networks of automata extended with clock and data variables. A simulator is a validation tool which enables examination of possible dynamic executions of a system during early design (or modeling) stages and thus provides an inexpensive mean of fault detection before verification by the model-checker which covers the exhaustive dynamic behavior of the system. The model-checker can check invariant and reachability properties by exploring the state-space of a system, i.e., reachability analysis concerning symbolic states represented by constraints.

Tools dedicated to the verification of security are for instance Cryptyc [96], Scyther [58], LySa [45] and Choreographer [90]. There are also tools targeting on a wide use and applicability to practical issues such as AVISPA tool suite [14], and AVANTSSAR platform [13]. AVISPA (automated validation of internet security protocols and applications) is a tool funded by the European Union, which provides a push-button, industrial-strength technology for the analysis of large-scale Internet security-sensitive protocols and applications. AVISPA uses several different model-checking approaches. Protocol models are written in the high-level protocol specification language (HLPSL). Protocols are specified in HLPSL regarding their roles, using control flow patterns, data structures, alternative adversary models, as well as different cryptographic primitives and their algebraic properties. HLPSL specification has a declarative semantics based on Lamport's temporal logic of actions[130] and an operational semantics defined regarding a rewrite-based formalism called the intermediate format. Once the model of the system is specified in HLPSL, AVISPA translates it into the intermediate format, which is an input format for AVISPA back-end model checkers.

AVISPA utilizes four back-end tools for validation of security protocols: On-the-fly model-checker (OFMC), constraint-logic-based attack searcher (CL-AtSe), SAT-based model-checker (SATMC), and tree automata based on automatic approximations for the analysis of security protocols (TA4SP). The advantage of having multiple back-ends is that only one model can be specified and it can be analyzed with four different tools.

AVANTSSAR (automated validation of trust and security of service-oriented architectures) is a follow-up project of AVISPA, introducing new languages for describing models, the AVANTSSAR specification languages ASLan++ and ASLan. ASLan++ [223] is a high-level formal language similar to the HLPSL, used for specifying security-sensitive service-oriented architectures, their associated security policies, and their trust and security properties. Translation formally defines the semantics of ASLan++ to ASLan, the low-level specification language that is the input language for the back-ends of the AVANTSSAR Platforms - OFMC, CL-AtSe, and SATMC:

- OFMC [32] combines many techniques to enable the efficient analysis of security properties. First, OFMC uses lazy data types as a simple way of building efficient on-the-fly model checkers for security properties with very large, or even infinite, state spaces. A lazy data type is one where data constructors build data without evaluating their arguments. Second, OFMC models the adversary in a lazy fashion, where adversary communication is represented symbolically and solved during a search. Third, while OFMC performs verification for a bounded number of sessions, it works with symbolic session generation, which avoids enumerating all possible ways of instantiating possible sessions. Fourth, OFMC exploits a state-space reduction technique, inspired by partial-order reduction, called constraint differentiation [164]. Constraint differentiation works by eliminating certain kinds of redundancies that arise in the search space when using constraints to represent and manipulate the messages that may be sent by the adversary. Finally, OFMC also provides some limited support for handling different equation specified operators on messages.

- Cl-Atse [216] represents protocol states symbolically as collections of non-ground facts, which record the states of different threads, the messages sent to the network, and the adversary knowledge. In particular, constraints are used to describe what the different agents know, and a constraint calculus is used to solve for what they can know, from messages previously exchanged, i.e., the calculus is used to solve a variant of the non-ground intruder deduction problem. CL-Atse was designed to allow the easy integration of new deduction rules and operator properties.

- SATMC [15] is an open platform for model checking of security services. SATMC reduces the problem of checking whether a protocol is vulnerable to attacks of bounded length to the satisfaction of ability of a propositional formula which is then solved by a state-of-the-art SAT solver. It is done by combining a reduction technique of protocol insecurity problems to planning problems and SAT-reduction techniques developed for planning and Lamport's Temporal Logic that allows for leveraging state-of-the-art SAT solvers. SATMC provides some distinguishing features, including the ability to check the protocol against complex temporal properties (e.g., fair exchange); analyze protocols (e.g., browser-based protocols) that assume messages are carried over secure channels.

Another approach is deductive verification [43]. It consists of generating from the system and its specifications (and possibly other annotations) a collection of mathematical *proof*

obligations, the truth of which imply conformance of the system to its specification, and discharging these obligations using either interactive or automatic theorem provers. This approach has the disadvantage that it typically requires the user to understand in detail why the system works correctly, and to convey this information to the verification system, either in the form of a sequence of theorems to be proved or in the form of specifications of system components (e.g. functions or procedures) and perhaps sub-components (such as loops or data structures).

7.2 Verification Tool Selection

Related to the topic of this thesis the Uppaal verification tool was chosen. The graphical user interface dedicated to describing the model in the form of the automaton (or more cooperating automata in the form of processes) seems satisfactory for the required purposes. The Uppaal is a toolbox for validation (via graphical simulation) and verification (via automatic model-checking) of real-time systems. It consists of two main parts: a graphical user interface and a model-checker engine. The graphical user interface is used for creating models for simulation and or verification. These models need to be specified in the format of Uppaal, that is described in this section with examples, and there are also mentioned the differences related to unified modeling language (UML) [69], that is considered as the standard modeling language.

The engine part of Uppaal tool is dedicated to verification, and it is by default executed on the same computer as the user interface, but can also run on a more powerful server. For this thesis, the same machine is used for creating models, simulation and also verification. Formal models presented in previous chapters can be defined in the format or language of Uppaal with some minor modifications. In order to provide this transformation from the formal definition of automata into the Uppaal format, the resulting format needs to be defined.

The idea is to model a system using timed automata [8], simulate it and then verify properties on it. Timed automata are finite state machines with time (clocks). The formal definition of timed automaton can be expressed as $TA = (L, l_0, C, A, E, I)$, where

- L is a set of locations,
- $l_0 \in L$ is the initial location,
- C is the set of clocks,
- A is the set of actions, co-actions, and internal τ-actions,
- $E \in L \times A \times B(C) \times 2^C \times L$ is a set of edges between locations with and action, a guard and a set of clocks to be reset,
- $I : L \to B(C)$ assigns invariants to locations.

A system consists of a network of processes that are composed of locations. Transitions between these locations define how the system behaves. The simulation step consists of running the system interactively to check that it works as intended. Then Uppaal can ask the verifier to check reachability properties, i.e., if a particular state is reachable or not. It is called model-checking, and it is an exhaustive search that covers all possible dynamic behaviors of the system. More precisely, the engine uses on-the-fly verification combined with

a symbolic technique reducing the verification problem to that of solving simple constraint systems [239, 134]. The verifier checks for simple invariants and reachability properties for efficiency reasons. Other properties may be checked by using testing automata [115] or the decorated system with debugging information [143].

The description language of Uppaal, which is based on graphical user interface, differs from standard UML representation of finite state automaton. Some major differences are presented in figure 7.1.

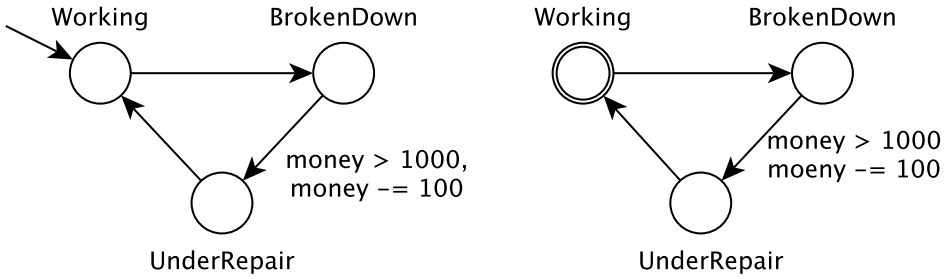

Figure 7.1: Example of Models with action

Note that these two models define the same automaton. The significant difference is the initial state, which is defined in UML (left image) in figure 7.1 depicted by the first arrow from no-where and on the right side of the same figure is the automaton in the Uppaal format with initial state marked as the location with double border. The double border is used in UML for the finite state, which is not explicitly defined in Uppaal tool. Finite state in Uppaal can be presented as a location from which does not exist any other transition to another state.

The transition between states can contain the condition which is called guard in Uppaal. When the condition is evaluated as positive (correct) the transition is enabled. Otherwise, this transition is not enabled, and the system remains in the same state. When a variable is part of a condition, it needs to be declared as a local variable of the state machine model or global variable of the system. Guards can restrict the possible state changes by disabling transitions. However, it can also extend the possible transition. In order to form more complex transitions, these actions can be assigned to the transition. An action and the transition are executed together. Actions are usually tided with a variable which updates.

A system in Uppaal is composed of concurrent processes, each of them modeled as an automaton. The automaton has a set of locations (states). Transitions are used to change location. To control when to take a transition (to "fire„ it), it is possible to have a guard and a synchronization. A guard is a condition on the variables and the clocks saying when the transition is enabled. There are two different types of synchronization: synchronization on *simple channel* or on *broadcast channel*. Both synchronizations require the declaration of the channel of the synchronization: message sending is realized on these channels. The synchronization mechanism in Uppaal is a hand-shaking synchronization: two processes take a transition at the same time, one will have an $a!$ and the other an $a?$, with a being the synchronization channel. When taking a transition, two actions are possible: assignment of variables or reset of clocks.

There is a synchronization on a single channel a only if there is a process (state machine) with an actual location from where there is an outgoing enabled edge (enabled transition) on which $a!$ is set, and there is the other process with an enabled transition on which $a?$ is set. This is depicted in figure 7.2. In the frames there are parts (locations) of the different processes, the top locations are the actual states of the processes. As far as the process in the left frame can send a synchronization message and the other process can receive it, the synchronization is enabled, and both transitions are executed together. However, if there were not a synchronization message sending transition or a receiver transition, then the synchronization would be disabled, and the transitions are also disabled.

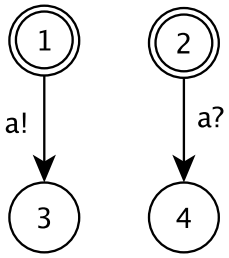

Figure 7.2: Simple synchronization example in Uppaal tool

If there are multiple receivers on the channel, the synchronization is executed only with one of them (chosen randomly). Broadcast synchronization happens between one sender and multiple receivers (amount of receivers could be zero and more). The receiver behaves similarly to the simple synchronization (if the transition is enabled and there is a synchronization message, the transition can fire). However, there is a difference from the sender point: sender can execute the transition with synchronization if there are multiple receivers and all of the receiver processes execute the synchronization transition.

7.3 Verification Models

Two formal model was defined in previous chapters, and these models need to be verified. More precisely, the model of implementation needs to be verified with the model of required behavior. However, these models were defined formally as Turing machine automata, and both have the second tape with the simulation of deterministic finite state automaton. According to the fact that both models have the same Turing machine, the verification process is focused on model checking of presented deterministic finite state automata. Turing machine was used in both cases in order to handle multiple files on a mobile device. For verification process is convenient to have a static amount of files which is not changing during the process. Moreover, Turing machines perform the same behavior for both models - model of required behavior and model of implementation. Since there are no differences between Turing machines, the verification of this automaton is omitted.

Note that proposed model of implementation is weaker than the precise model. Therefore more vulnerabilities can be found, and it is expected behavior. In order to be more precise, the model can be adjusted, and a process of verification can be started again.

Definition of models is provided in the format of Uppaal tool, which was already described in this chapter. Models are verified with a user process. The user process is another

finite state machine, which is non-deterministic. A user is simulated by this machine, which sends commands to both models. These commands are depicted randomly by the definition of non-deterministic transition by Uppaal. For the thesis, three types of user are created. The first one is a general user, which sends a sequence of all possible operations defined by expression 7.1 to both models (model of required behavior and model of implementation). The second one is limited to work with public files only (sends operations which are related to public file only). The last one is the same but sends operation limited to private file usage.

$$operations = \{open\_public, open\_private, read\_public, read\_private, write\_public, write\_private, share\_content, seek\_position, copy, close\_public, close\_private\} \quad (7.1)$$

Commands Declaration

In order to verify working with a sequence of commands defined in expression 7.1, there needs to exist declaration of these commands. Unfortunately, Uppaal does not support data types such as *string* or array of characters. Therefore, the set of commands can be encoded into numbers. Each command has its constant value. This value can be then assigned to share variable called *command*. This command variable has initially zero value, which does not belong to any command.

Example of declaration these commands is described in listing 7.1.

```
// shared command variable
int command = 0;

// constants of command type
const int open_public   = 1;
const int open_private  = 2;
const int read_public   = 3;
const int read_private  = 4;
const int write_public  = 5;
const int write_private = 6;
const int share_content = 7;
const int seek_position = 8;
const int copy          = 9;
const int close_public  = 10;
const int close_private = 11;
```

Listing 7.1: Declaration of variables for verification

Synchronization Channels

To set required behavior in Uppaal tool mechanism for synchronization was defined. These two mechanisms are finite state automaton, which is waiting for the synchronization command and the fire another synchronization signal through a different channel. The reason for that is the update of the variable is provided as the last part of the transition. For instance lets have two states and one transition with condition that $a == 3$, waiting for channel $a?$, and with update the variable $a = 4$. The evaluation is defined as when the signal

arrives from channel a, and the condition is evaluated as true, then the update is performed. In order to have two processes (state machines) synchronized with message passing value through shared variable, it is required to define another process (state machine).

The definition of channels is depicted on listing 7.2. Declaration of channels, constants, and the shared variable command is defined on the global level. The reason is that each process (each model of automaton) can access its values and the synchronization is provided for the whole system.

```
// synchronization channels
chan read_command;
chan next_command;
chan user_action;
broadcast chan read;
```

Listing 7.2: Declaration of channels for verification

Unfortunately Uppaal tool does not support value passing through the channels [36], but this can be simulated by shared variable and few synchronization mechanism. Note that it is not clean to do *read!*, $x = 3$ and *read?*, $y = x$, where *read* is a channel and x, y are variables.

For this thesis, two synchronization processes are needed. The first one is situated between user model (definition follows in this section) and shared command variable, which holds the message. Note that the user sends commands to the model of required behavior and also to the model of implementation via the shared command variable. The user does not read the value. It is one direction flow of information. When the user sends its command, there is need to be written, and afterward, these models can read the value. The mechanism of the synchronization can be defined as two state finite state automaton, called as the Write process illustrated in figure 7.3.

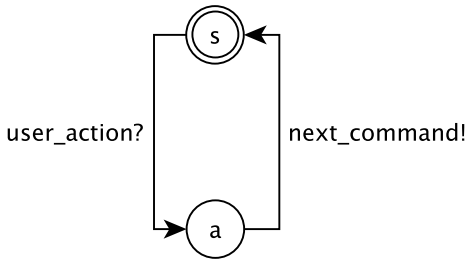

Figure 7.3: Write process synchronization model

Figure 7.3 describe the automaton which is waiting for a signal from the channel called *user_action?* that is sent from user model. After that, the transition from the initial location is performed. After that, the second transition is executed, and the signal is sent through the channel *next_command*.

Similar automaton is used for reading a value from the shared variable command. The Reader process also has two states with transition related to channel reaction only. This process waits for new value in the command variable and the event from the Write process. When event through *next_command* channel arrives, and the next step is produced,

the broadcast channel *read* fire the event to the model of required behavior, the model of implementation, and to the user model. The first two models read the value an if is it possible perform the transition between their states. The user model just goes back to its initial state and can produce next command.

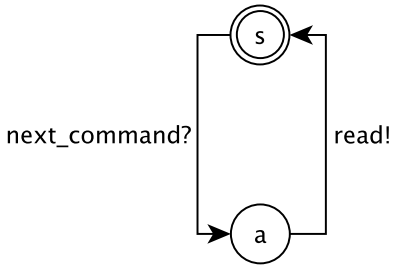

Figure 7.4: Reader process synchronization model

Figure 7.4 shows the automaton for synchronizing the reading of shared variable *command*. The automaton waits for a signal on channel *next_command?* that is sent from the Writer process. Afterwards, the transition from the initial state is performed. The next transition in this process sends the signal through the channel *read!* to subscribers of this channel.

Model of User

A user is unpredictable part of the verification process, and therefore it is defined as a non-deterministic automaton. The reason for non-determinism is that the transition is chosen randomly by the Uppaal verification engine. There is no defined pattern of the sequence of commands that the user should provide. The set of all available commands was defined by expression 7.1 and the general user model should be able to send each command to waiting models (model of required behavior and model of implementation).

More precisely, the user model should provide the step of its automaton, and the result writes into the shared variable command. Then synchronization processes perform their tasks. The general user model is illustrated in figure 7.5. Note that there appear two lines on some transitions, the reason is that the first line defines the update of shared variable *command* during the transition and also sending the signal on the channel *user_action* to its listeners.

The user model starts its execution in *s* state. Since there is no guard on any transition from state *s* the Uppaal chose randomly which transition will be done. Each transition from state *s* is defined as sending the signal through channel *user_action*. The receiver of this channel is the Writer process. The result of the transition is the change of shared variable *command*.

After the successful transition into any state except the state *s* the execution of this automaton is blocked until the signal from channel *read* arrives. The *read* channel synchronize the reading of the value of shared variable in all processes.

The model of behavior and also the model of implementation cannot provide their first transition without the decision of file type (public or private). The synchronization processes (Writer and Reader) were defined, and user model (user process) provides the behavior of a prospective user. These processes are necessary for performing model checking

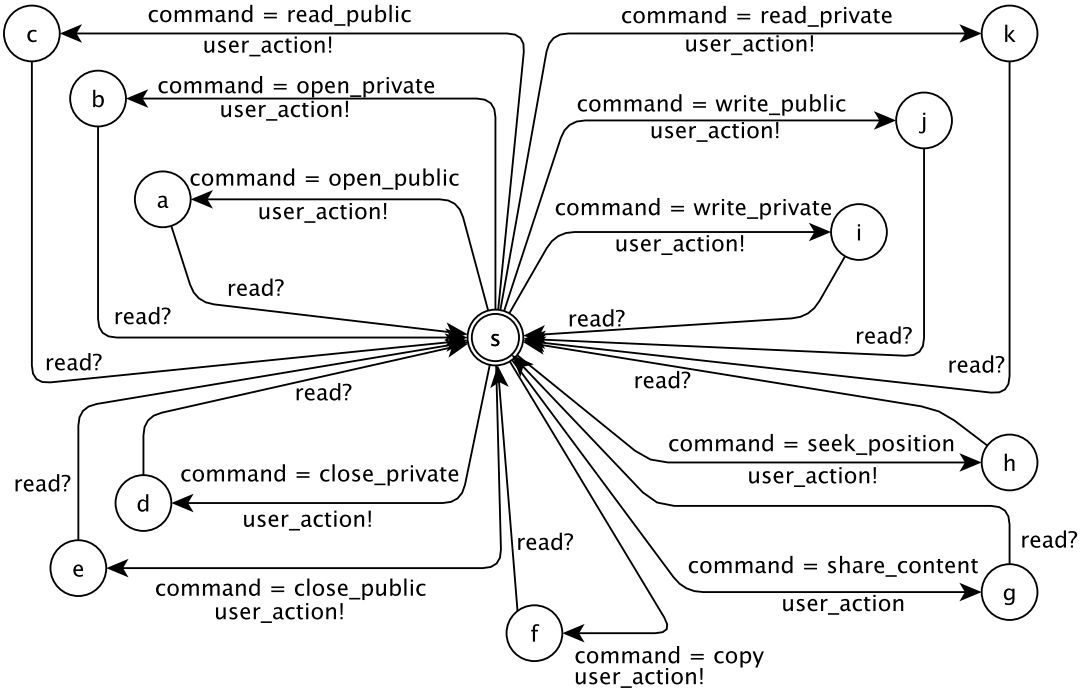

Figure 7.5: General user nodel for verification

on two formal models defined in previous chapters. The models are defined in the Uppaal format in the following sections.

Verification Model of Required Behavior

A model which defines required behavior was defined in the formal format consisting of Turing machine for handling multiple files on the mobile device and the decision logic formally defined as finite state automaton. This automaton is described in the Uppaal format and marked as the required behavior model. The model is illustrated in figure 7.6.

Note that the model is equal to its formal definition. The evident difference is that the Uppaal model does not have finite states. This model is modeled as a never-ending finite state machine. This aspect cannot be considered as the wrong model. Moreover, this difference does not have any impact on the verification results. The model has on its transitions one guard (condition), which can be identified by the symbol of equality (==) and communication with synchronization channel *read?*. Some transitions can be done via more than one specific command.

Therefore more choices are defined by OR symbol (||). The synchronization channel *read* is waiting for the Reading process, which informs about new command presented in the shared variable *command*.

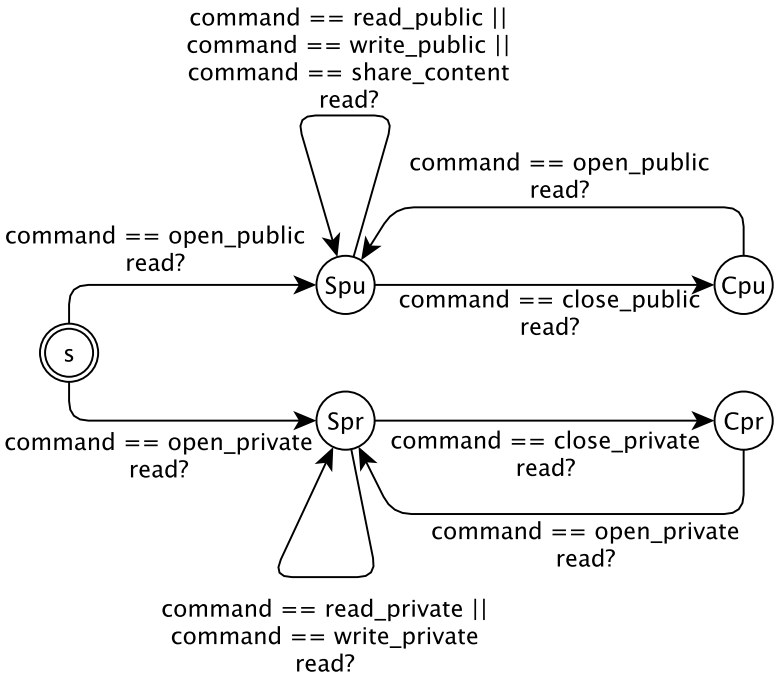

Figure 7.6: Model of required behavior in Uppaal format

Verification Model of Implementation

A formal model of implementation has the same feature with Turing machine. Moreover, the Turing machine has the same behavior as the model of required behavior has. The main implementation logic is defined as finite state automaton, and this logic is also presented in the Uppaal format in figure 7.7. File operations are mainly defined in the states *Wpu* and *Wpr*.

These two states operate not on file level, but with memory, that is tainted, and the flow of data is handled in the implementation. The model of implementation in Uppaal format does not have a finite state, as was already described earlier.

States *Cpu* and *Cpr* are finite in the formal model and the reason is that during these states opened file is released from memory. However, users usually do not close file manually, but they close the application itself which perform closing operation on behalf of a user. Model in Uppaal tool is defined as a process which is opened application with a file, and the life-cycle is never ended. The verification process is not focused on the finite states, but to the whole behavior of model related to required behavior.

7.4 Summary

This chapter introduces verification process as the measurement of the checking the properties represented as primary states between the model of required behavior and model of implementation. During this chapter, the verification tools were described, and Uppaal was chosen as an appropriate tool for model checking in order to verify proposed solution with

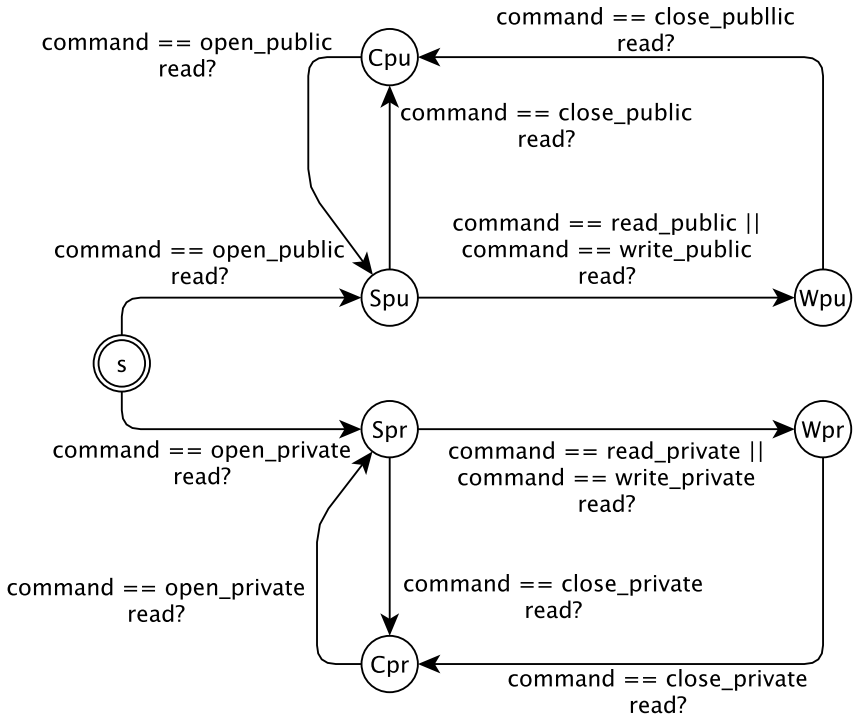

Figure 7.7: Model of implementation in Uppaal format

the model of required behavior. The implementation details of verification process with the variables and command declarations were described, and source code presented.

Moreover, the models were specified in the format of the Uppaal and synchronization processes needed for verification process were also described. These synchronization processes were used for simplifying the message passing between a user and the models. In order to define the user, the additional model was introduced. This model performs sending the commands to both checking models.

The next chapter provides verification experiments with these models inside Uppaal verification tool.

Chapter 8

Verification Experiments

This chapter introduces verification experiments based on models, which are processes in the verification tool. Models were defined by the graphical user interface and then simulated. The next phase of confirmation that the implementation solution model satisfy required behavior is called verification. Verification is focused on user actions that are sends to both models (model of required behavior and model of implementation). Result of the verification process is the report consisting of required behavior defined by formulas defined in this chapter and their results provided by Uppaal engine.

Aim of the verification experiments is confirmation that the implementation satisfy required behavior. Otherwise the Uppaal engine should find the counterexample. When counterexample is found, the discussion about that occurrence is provided. Since the model of implementation in any rule does not satisfy the required behavior does not necessarily means that the implementation is wrong.

There are two possible explanation before the experiments starts. The first case can be that the model of implementation is simplified and does not cover the whole functionality of implementation. In this case the model can be justified or updated in the specific sections, that does not fulfill the required behavior. The second reason can be identified on the side of model of required behavior, which for example does not provide transition for specific user command. In this case the model of required behavior has different state than the model of implementation. The results of each experiment in this chapter are discussed with attention to details, when any verification rule does not confirm required behavior.

Verification experiments are based on Uppaal query language [36], which is based on time computational tree logig (TCTL) quantifiers [95]. In very short description, the queries available in the Uppal verifier engine are:

- **E<> p**: there exists a path where p eventually holds.
- **A[] p**: for all paths p always hold.
- **E[] p**: there exists a path where p always holds.
- **A<> p**: for all paths p will eventually hold.
- **p –> q**: whenever p holds q will eventually holds.

where p and q are state formulas. For example formula *P1.cs* means that the process (a state machine) P is in the state *cs*. The full grammar of the query language is available in the on-line help of Uppaal tool. Moreover Uppal verification tool provides verification

for these properties: reachability, safety and liveness property, which uses the previously defined queries. During this chapter three examples of verification are presented. These verification examples should provide introduction into verification of two models, in which the user provides sequence of commands to both models and these models should be in consistent state. The consistency is checked by properties defined in Uppaal query language.

Related to definition of synchronization models, user model, and both models that are verified (model of required behavior and model of implementation), the following sections uses the terms model and process in the same meaning. The reason is that the formal model presented in Uppaal format is during verification process transformed into process.

8.1 Experiment 1

The first verification experiments is focused on the basic model checking which relates between states of the required behavior automaton and implementation automaton. The verification is dedicated to check if both models are in the same states when user sends commands (file operations) into application. The experiment uses user model as was defined in previous section, and it is called *UserProcess* B.4.

Moreover, model of required behavior is named *FSMRequiredProcess* B.2, model of implementation is called *FSMImplementationProcess* B.3. In addition, the verification process have synchronization processes presented earlier. Each state machine is defined as process in the Uppaal verification tool.

In order to check the consistency of initial states of automata, the rules depicted on listing 8.1 are used. In other words this mean that the model of required behavior (FSMRequiredProcess) is in the same initial state as the model of implementation (FSMImplementationProcess).

```
A[] FSMRequiredProcess.s imply FSMImplementationProcess.s

A[] FSMImplementationProcess.s imply FSMRequiredProcess.s
```

Listing 8.1: Verification rule for initial states

Next rules depicted in listing 8.2 verify the states related to opened file. Since the file is open as public in the model of required behavior it is not possible to have the same file opened as public in the model of implementation and vice versa. In details, when the model of required behavior is in state *Spu* (working with public file), the model of implementation should not be in the state *Spr* or *Wpr* (working with private file). The same should be valid for working with private file in model of required behavior with state *Spr* and model of implementation and states *Spu* or *Wpu*.

```
A[] not (FSMRequiredProcess.Spu and
    (FSMImplementationProcess.Spr or FSMImplementationProcess.Wpr))

A[] not ((FSMImplementationProcess.Spr or FSMImplementationProcess.Wpr)
    and (FSMRequiredProcess.Spu))

A[] not (FSMRequiredProcess.Spr and
    (FSMImplementationProcess.Spu or FSMImplementationProcess.Wpu))
```

```
A[] not ((FSMImplementationProcess.Spu or FSMImplementationProcess.Wpu)
    imply (FSMRequiredProcess.Spr))
```

Listing 8.2: Verification rules for opened file

In addition, working state of file in model of required behavior is named as *Spu* or *Spr* (according to file category). The same state is expected in the state of implementation, thus next two rules are presented. Related to implementation details (and also formal model) the working status for file can be selected in one of the possible combination of states *Spu* and *Wpu* or *Spr* and *Wpr*. These properties are verified with following rules presented in listing 8.3.

```
A[] FSMRequiredProcess.Spu imply
    (FSMImplementationProcess.Spu or FSMImplementationProcess.Wpu)

A[] (FSMImplementationProcess.Spu or FSMImplementationProcess.Wpu)
    imply FSMRequiredProcess.Spu

A[] FSMRequiredProcess.Spr imply
    (FSMImplementationProcess.Spr or FSMImplementationProcess.Wpr)

A[] (FSMImplementationProcess.Spr or FSMImplementationProcess.Wpr)
    imply FSMRequiredProcess.Spr
```

Listing 8.3: Verification rules for working states of file

Another part of this experiment is focused on closing file operation. The opened file should be always closed by the user, otherwise the system will close the file when the application is closed. In the formal definition of both models, the user is able to close the file with the operation related to category of the file. Private file should be closed by the operation *close_private* and public file with operation *close_public*. Related to this fact and formal definition of models, model of required behavior should be in one of these states *Cpu* or *Cpr* (according to open file category) and the model of implementation has to be in the same state, because the verification works with both models in the same manner. Verification rules expressed in listing 8.4 also check for correct file operation (*command* sent by the user model).

```
A[] FSMRequiredProcess.Cpu and FSMImplementationProcess.Cpu

A[] FSMRequiredProcess.Cpr and FSMImplementationProcess.Cpr

A[] (FSMRequiredProcess.Cpu and command == close_public)
    imply (FSMImplementationProcess.Cpu and command == close_public)

A[] (FSMRequiredProcess.Cpr and command == close_private)
    imply (FSMImplementationProcess.Cpr and command == close_private)
```

Listing 8.4: Verification rules for closing file

Experiment 1 Results

The rules mentioned in the previous section were collected together and were run on the system consisting of all related processes (models and their synchronization models). This was focused on basic states transition and the distinguish between public and private file in both models. For the purpose of general verification the user model with the ability of random choosing command with the meaning of file operation was used. In order to provide the process of verification through all possible combination of command this user model seems the right choice. The consistency was verified through the states related to opening and closing file and also for working statuses. The closing file was also verified through the checking of command value sent by user model. The results presented in listing 8.5 of the experiment is described as log file from Uppaal tool.

```
A[] FSMRequiredProcess.s imply FSMImplementationProcess.s
Property is satisfied.

A[] FSMImplementationProcess.s imply FSMRequiredProcess.s
Property is satisfied.

A[] not (FSMRequiredProcess.Spu and
    (FSMImplementationProcess.Spr or FSMImplementationProcess.Wpr))
Property is satisfied.

A[] not ((FSMImplementationProcess.Spr or FSMImplementationProcess.Wpr)
    and (FSMRequiredProcess.Spu))
Property is satisfied.

A[] not (FSMRequiredProcess.Spr and
    (FSMImplementationProcess.Spu or FSMImplementationProcess.Wpu))
Property is satisfied.

A[] not ((FSMImplementationProcess.Spu or FSMImplementationProcess.Wpu)
    imply (FSMRequiredProcess.Spr))
Property is satisfied.

A[] FSMRequiredProcess.Spu imply
    (FSMImplementationProcess.Spu or FSMImplementationProcess.Wpu)
Property is satisfied.

A[] (FSMImplementationProcess.Spu or FSMImplementationProcess.Wpu)
    imply FSMRequiredProcess.Spu
Property is satisfied.

A[] FSMRequiredProcess.Spr imply
    (FSMImplementationProcess.Spr or FSMImplementationProcess.Wpr)
Property is satisfied.

A[] (FSMImplementationProcess.Spr or FSMImplementationProcess.Wpr)
    imply FSMRequiredProcess.Spr
Property is satisfied.

A[] FSMRequiredProcess.Cpu and FSMImplementationProcess.Cpu
Property is satisfied.

A[] FSMRequiredProcess.Cpr and FSMImplementationProcess.Cpr
Property is satisfied.
```

```
A[] (FSMRequiredProcess.Cpu and command == close_public)
    imply (FSMImplementationProcess.Cpu and command == close_public)
Property is satisfied.

A[] (FSMRequiredProcess.Cpr and command == close_private)
    imply (FSMImplementationProcess.Cpr and command == close_private)
Property is satisfied.
```

Listing 8.5: Results of verification experiment 1

This experiment shows that the general required behavior of implementation method should be satisfied. The results should not be considered that the model of implementation is perfectly correct, the reason is that the verification of all possible combination of states were not checked and also the model of implementation is defined from the higher perspective and some details are omitted. In order to be more precise the model can be adjusted.

8.2 Experiment 2

This experiment is focused on the verification of the same models as the previous experiment. Moreover, this experiment is limited to private file operations only. There should be checked that the model of implementation does not provide any space for data leakage through available file operations, system operations, or other available features. For the purpose of this verification the general user process was modified. Basically the same models are verified and the presence of synchronization models are required.

In order to be sure, that the required part of verified models will be used, which means the subset of available commands related to the private files is considered in this experiment. User model needs to be modified in order to provide at least the control transition at the beginning of the verification process. The control transition is the only one possible command that is send to checking models and this command is *open_private*.

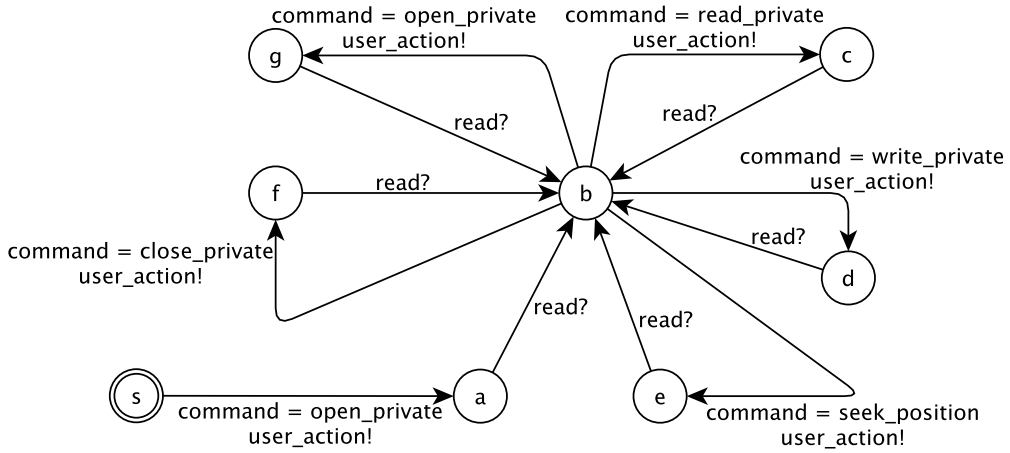

Figure 8.1: User model for private file operations in Uppaal format

For performance purposes and the requirement for control transitions on the user process, a set of allowed file operations were limited to union of sets of available transition operations defined by *FSMRequiredProcess* (model of required behavior) and *FSMImplementationProcess* (model of implementation). This approach removes operations with public files, which in this case perform the waiting of these checking automata, because there is not defined the transition for this kind of operations. Updated model of user is defined in Uppaal format and illustrated in figure 8.1. One type of transitions defined on this model consist of two lines, which denotes updating of share variable *command* and sending the signal to listeners of synchronization channel *user_action*. The second type of transition is the waiting for the signal on channel *read* until the content of the variable *command* is read. Source code of user process focused on private file operations is defined in listing B.6.

According to figure 8.1, the initial state is *s* and the only available transition is via the command *open_private* as was already described. The next is the synchronization transition, which checks that the command value has been read. Following transitions are randomly selected as in the general user model used in the Experiment 1. The set of rules from Experiment 1 is still valid even for this limited user model.

The rules that verify the reachability of states related to opening or working with public files are presented on listing 8.6. Rules are specified as provide satisfied result when these states are not possible to reach. The formula can be also presented in the opposite way, for instance the result is considered as satisfied when required states are reached via any transition.

```
A[]] not (FSMRequiredProcess.Spu or FSMImplementationProcess.Spu or
    FSMImplementationProcess.Wpu)

A[]] not (FSMRequiredProcess.Cpu or FSMImplementationProcess.Cpu)
```

Listing 8.6: Verification rules for reachability states related to public file operations

The reachability of the state *Spr* in both checking models (model of required behavior and model of implementation) is defined by the transition with the only one command *open_private*. This consistency is checked by the rule expressed in listing 8.7. From this state the file operations can be performed on the file. The reason is that in this point the existing file is opened (or new one will be created in near future with file operation *write_private*).

```
A<> (FSMRequiredProcess.Spr and command == open_private)
    imply (FSMImplementationProcess.Spr and command == open_private)

A<> (FSMImplementationProcess.Spr and command == open_private)
    imply (FSMRequiredProcess.Spr and command == open_private)

A<> not ((FSMRequiredProcess.Spr and command == open_public)
    imply (FSMImplementationProcess.Spr and command == open_public))

A<> not ((FSMImplementationProcess.Spr and command == open_public)
    imply (FSMRequiredProcess.Spr and command == open_public))
```

Listing 8.7: Verification rules for reachability *Spr* state

The model of required behavior defines available transitions on already opened file as *read_private* and *write_private* and the automaton persist in the same state *Spr*. In order to provide any file operations which can be taint by the prototype the model of required behavior has its own state for working with the file. This difference should not be considered as malfunction or any other kind of vulnerability. However, the amount of all possible operations should be verified with these rules presented in listing 8.8.

```
A<> (FSMRequiredProcess.Spr and command == read_private)
    imply (FSMImplementationProcess.Wpr and command == read_private)

A<> (FSMImplementationProcess.Wpr and command == read_private)
    or (FSMImplementationProcess.Spr and command = read_private)

A<> (FSMRequiredProcess.Spr and command == write_private)
    imply (FSMImplementationProcess.Wpr and command == write_private)

A<> (FSMImplementationProcess.Wpr and command == write_private)
    imply (FSMRequiredProcess.Spr and command == write_private)

A<> (FSMRequiredProcess.Spr and command == seek_position)
    imply (FSMImplementationProcess.Wpr and command == seek_position)

A<> (FSMImplementationProcess.Wpr and command == seek_position)
    imply (FSMRequiredProcess.Spr and command == seek_position)
```

Listing 8.8: Verification rules for file operations during working with file

There are no more possible transition commands available. Commands related to closing file were already tested during the experiment 1.

Experiment 2 Results

Results expressed in listing 8.9 of this experiment are presented in the form of log from Uppaal tool in the same format as in the Experiment 1. This experiment was focused on transition logic between states and checking models. These states and transitions defined in the model of required behavior should be consistent with the model of implementation. Model of implementation has different state for working with file, the reason is checking the file content as was presented in the prototype. However, there are differences in models, the protection of private files should be preserved.

```
A[] not (FSMRequiredProcess.Spu or FSMImplementationProcess.Spu or
    FSMImplementationProcess.Wpu)
Property is satisfied.

A[] not (FSMRequiredProcess.Cpu or FSMImplementationProcess.Cpu)
Property is satisfied.

A<> (FSMRequiredProcess.Spr and command == open_private)
    imply (FSMImplementationProcess.Spr and command == open_private)
Property is satisfied.

A<> (FSMImplementationProcess.Spr and command == open_private)
    imply (FSMRequiredProcess.Spr and command == open_private)
```

```
Property is satisfied.

A<> not ((FSMRequiredProcess.Spr and command == open_public)
    imply (FSMImplementationProcess.Spr and command == open_public))
Property is satisfied.

A<> not ((FSMImplementationProcess.Spr and command == open_public)
    imply (FSMRequiredProcess.Spr and command == open_public))
Property is satisfied.

A<> (FSMRequiredProcess.Spr and command == read_private)
    imply (FSMImplementationProcess.Wpr and command == read_private)
Property is satisfied.

A<> (FSMImplementationProcess.Wpr and command == read_private)
    or (FSMImplementationProcess.Spr and command = read_private)
Property is satisfied.

A<> (FSMRequiredProcess.Spr and command == write_private)
    imply (FSMImplementationProcess.Wpr and command == write_private)
Property is satisfied.

A<> (FSMImplementationProcess.Wpr and command == write_private)
    imply (FSMRequiredProcess.Spr and command == write_private)
Property is satisfied.

A<> (FSMRequiredProcess.Spr and command == seek_position)
    imply (FSMImplementationProcess.Wpr and command == seek_position)
Property is satisfied.

A<> (FSMImplementationProcess.Wpr and command == seek_position)
    imply (FSMRequiredProcess.Spr and command == seek_position)
Property is not satisfied.
```

Listing 8.9: Results of verification experiment 2

According to results depicted in listing 8.9, the last rule is not satisfied. The reason is the the command *seek_position* is not defined in the model of required behavior and the automaton can not provide transition. Seek operation defined on file is defined as moving the position in already opened file, which is used for reading or writing from/to the file. This file operation is not considered as vulnerability in the purpose of the thesis. There are two possible ways in order to satisfy the last rule. The first approach is to adjust the model of required behavior with the adding the missing file operation into corresponding transition and the second one is to remove the operation from the model of implementation, and also update the implementation solution in order to deny this system function call. Note that, related to implementation of the prototype, this system call was not appeared during working with prototype.

8.3 Experiment 3

The last experiment related to verification is focused on the public file operations. General requirements targeted on participants are the same as in previously described experiments, in other words the model of required behavior, model of implementation, user model and synchronization models are mandatory for this experiment. The user model is updated for

this type of experiment in very similar way as the user model is modified in the experiment 2.

The current user model has control over the initial state and the first transition with command equal to *open_public* and than other commands are chosen randomly. The illustration in figure 8.2 describe the modification of general user model. During transition can be updated the share variable *command* and the signal for waiting processes *user_action*. The second type of transition is waiting for signal on the synchronization channel *read*.

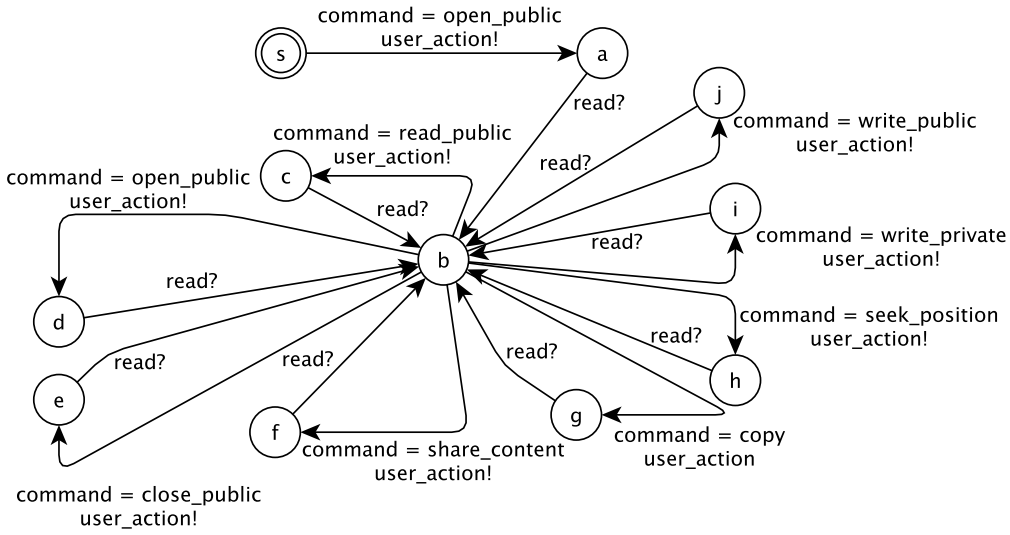

Figure 8.2: User model for public file operations in Uppaal format

Figure 8.2 with automaton begins its execution in the state s and generates the user action with command *open_public*, as is required for this experiment. When the signal with this command is confirmed and read, there are all possible commands that appear on the public part of the model of implementation and model of required behavior. Model checking is based on checking of all possible commands that are send to the verified models.

This experiment is focused on working with public files and their ability to influence private files. There should not be possible to set opened public file as private. In the words of automaton, there should not be a path from public states of automaton (Spu, Wpu, and Cpu) into private states of the same automaton (Spr, Wpr, and Cpr). Moreover, there should be also consistency between model of required behavior and model of implementation. Rules for checking reachability properties for private states of automaton, which use updated user model with public file operations, is described on listing 8.10. Source code of user process focused on public file operations is defined in listing B.5.

```
E<> not (FSMRequiredProcess.Spr or FSMImplementationProcess.Spr or
    FSMImplementationProcess.Wpr)

E<> not (FSMRequiredProcess.Cpr or FSMImplementationProcess.Cpr)
```

Listing 8.10: Verification rules for reachability of private states

These rules describe two three private states that should not be possible to reach via available commands. In addition, when the automata (*FSMRequiredProcess* and *FSMImplementationProcess*) are in the state *Spu*, it is not possible to reach private states even thought the general user model is used. Related to modified user model, the accessibility of state *Spr* needs to be verified in order to cover the correct opening of the private file. The rules presented on listing 8.11 checks for this property.

```
A<> (FSMRequiredProcess.Spu and command == open_public)
    imply (FSMImplementationProcess.Spu and command == open_public)

A<> (FSMImplementationProcess.Spu and command == open_public)
    imply (FSMRequiredProcess.Spu and command == open_public)

A<> not ((FSMRequiredProcess.Spu and command == open_private)
    imply (FSMImplementationProcess.Spu and command == open_private))

A<> not ((FSMImplementationProcess.Spu and command == open_private)
    imply (FSMRequiredProcess.Spu and command == open_private))
```

Listing 8.11: Verification rules for accessibility the state *Spr*

The next verification rules presented in listing 8.12 check for consistency of operations related to public file, such as *read_public*, *write_public* and *share_content*. These operations are available in both models and should perform safe file operations.

```
A<> (FSMRequiredProcess.Spu and command == read_public)
    imply (FSMImplementationProcess.Wpu and command == read_public)

A<> (FSMImplementationProcess.Wpu and command == read_public)
    imply (FSMRequiredProcess.Spu and command == read_public)

A<> (FSMRequiredProcess.Spu and command == write_public)
    imply (FSMImplementationProcess.Wpu and command == write_public)

A<> (FSMImplementationProcess.Wpu and command == write_public)
    imply (FSMRequiredProcess.Spu and command == write_public)

A<> (FSMRequiredProcess.Spu and command == share_content)
    imply (FSMImplementationProcess.Wpu and command == share_content)

A<> (FSMImplementationProcess.Wpu and command == share_content)
    imply (FSMRequiredProcess.Spu and command == share_content)
```

Listing 8.12: Verification rules for public file operations

The model of implementation defines three additional transition compared to model of required behavior and these transitions should be also verified. Verification rules for these properties are depicted on listing 8.13. The first transition command *seek_position* was already discussed in the previous experiment. This system call function can be considered as safe, because it just define the position inside the file. Another specific transition related to model of implementation is called *copy*, this is special case of sharing data, which is not available as feature for private file operations.

```
A<> (FSMImplementationProcess.Wpu and command == seek_position)
    imply (FSMRequiredProcess.Spu and command == seek_position)

A<> (FSMRequiredProcess.Spu and command == seek_position)
    imply (FSMImplementationProcess.Wpu and command == seek_position)

A<> (FSMImplementationProcess.Wpu and command == copy)
    imply (FSMRequiredProcess.Spu and command == copy)

A<> (FSMRequiredProcess.Spu and command == copy)
    imply (FSMImplementationProcess.Wpu and command == copy)

A<> (FSMImplementationProcess.Wpu and command == write_private)
    imply (FSMRequiredProcess.Spu and command == write_private)

A<> (FSMRequiredPRocess.Spu and command == write_private)
    imply (FSMImplementationProcess.Wpu and command == write_private)
```

Listing 8.13: Verification rules for additional implementation public file operations

In order to correctly handle this situation the specific mechanism was implemented and described in chapter with prototype. The last different transition *write_private* can influence private files. This transition defines that a content of any public file can be saved as private file. From the security point of view this property limits the available operations and does not provide any security breach or possible data leakage. Moreover, there is a possibility of updating private file, this case is under control of implementation.

Experiment 3 Results

Presented rules were collected and used in Uppal verification tool, the results are expressed on listing 8.14. There are more unsatisfied rules then in previous experiments. However, there are differences between model of required behavior and model of implementation and results confirms this fact, the content of files should be protected and private file content should not be shared or copied. Implementation provides the ability to update already opened private file with content of opened public file, this can be expressed by the transition on public file with command *write_private*. This behavior is not explicitly described by the model of required behavior. The implementation provides the ability to write the public content of the memory into the private file, but with the restriction focused on append mode.

```
E<> not (FSMRequiredProcess.Spr or FSMImplementationProcess.Spr or
    FSMImplementationProcess.Wpr)
Property is satisfied.

E<> not (FSMRequiredProcess.Cpr or FSMImplementationProcess.Cpr)
Property is satisfied.

A<> (FSMRequiredProcess.Spu and command == open_public)
    imply (FSMImplementationProcess.Spu and command == open_public)
Property is satisfied.

A<> (FSMImplementationProcess.Spu and command == open_public)
    imply (FSMRequiredProcess.Spu and command == open_public)
Property is satisfied.
```

```
A<> not ((FSMRequiredProcess.Spu and command == open_private)
    imply (FSMImplementationProcess.Spu and command == open_private))
Property is satisfied.

A<> not ((FSMImplementationProcess.Spu and command == open_private)
    imply (FSMRequiredProcess.Spu and command == open_private))
Property is satisfied.

A<> (FSMRequiredProcess.Spu and command == read_public)
    imply (FSMImplementationProcess.Wpu and command == read_public)
Property is satisfied.

A<> (FSMImplementationProcess.Wpu and command == read_public)
    imply (FSMRequiredProcess.Spu and command == read_public)
Property is satisfied.

A<> (FSMRequiredProcess.Spu and command == write_public)
    imply (FSMImplementationProcess.Wpu and command == write_public)
Property is satisfied.

A<> (FSMImplementationProcess.Wpu and command == write_public)
    imply (FSMRequiredProcess.Spu and command == write_public)
Property is satisfied.

A<> (FSMRequiredProcess.Spu and command == share_content)
    imply (FSMImplementationProcess.Wpu and command == share_content)
Property is satisfied.

A<> (FSMImplementationProcess.Wpu and command == share_content)
    imply (FSMRequiredProcess.Spu and command == share_content)
Property is not satisfied.

A<> (FSMImplementationProcess.Wpu and command == seek_position)
    imply (FSMRequiredProcess.Spu and command == seek_position)
Property is not satisfied.

A<> (FSMRequiredProcess.Spu and command == seek_position)
    imply (FSMImplementationProcess.Wpu and command == seek_position)
Property is satisfied.

A<> (FSMImplementationProcess.Wpu and command == copy)
    imply (FSMRequiredProcess.Spu and command == copy)
Property is not satisfied.

A<> (FSMRequiredProcess.Spu and command == copy)
    imply (FSMImplementationProcess.Wpu and command == copy)
Property is satisfied.

A<> (FSMImplementationProcess.Wpu and command == write_private)
    imply (FSMRequiredProcess.Spu and command == write_private)
Property is not satisfied.

A<> (FSMRequiredProcess.Spu and command == write_private)
    imply (FSMImplementationProcess.Wpu and command == write_private)
Property is satisfied.
```

Listing 8.14: Results of verification experiment 3

Rules which were not satisfied are related to the transitions with commands, that are not defined in the model of required behavior. The model of required behavior can be adjusted to handle these missing file operations. In contrast, the model of implementation can be more strict and some operations can be removed from model and restricted in its implementation in order to have consistency.

8.4 Summary

This chapter presented verification experiments, which were verified by Uppaal verification tool. Models (model of required behavior and model of implementation) described in the Uppaal format were verified via sending commands by the User model. User model was split into three categories - general user model, public user model and private user model. Each experiment uses different user model. Verified properties were focused on consistency of states between checking models and their transition commands. Next chapter provides experiments related to implementation solution.

Verification goals were defined with the Uppaal Query Language and within this language security properties are defined, but not explicitly. This thesis is focused on data protection with the ability of avoidance data leakage from the device. Results of the Experiment 1 described that there is not possible to provide file operations from different category on already opened file. In other words, for public file it is not possible to use file operations dedicated to private file and vice versa.

The Experiment 2 was focused on the private file category and Experiment 3 on the public file category. Both experiments shows that there is limited set of available operations, that can be performed on each file. Experiments shows that the model of implementation should satisfy the basic requirements defined by model of required behavior.

Experiments shows that the state of both checking automata are in consistent states, which means that content of private file should be kept on the device. Related to to model of implementation, there is no transition defined for sharing, copying or writing the content of private file into other place than the private file itself. However, the results of experiments were almost satisfied, it does not mean that the implementation solution is without errors or completely without vulnerabilities. The model of implementation was simplified in order to provide the results in convenient time and also in order to avoid state space explosion. For the purpose of the thesis the model should be considered as sufficient.

Chapter 9

Implementation Experiments

The theoretical part is covered, and the prototype was implemented, and this chapter introduces the experiments on randomly downloaded Android applications and the applying restriction in the way of limitation of permission enforcement related to open input files. The selection of the application was made to handle input in the file format. For instance, it does not make any sense to modify application which is not related to file manipulation such as game-based applications, internet browser, an alarm or any other kind of applications without a file on the input.

There are evaluated results from intercepted system calls with the obtained theoretical knowledge, described in earlier. Experiments are concerned with the assembly testing with the real selected applications such as *IO File Manager* (application 1) or *Ted Text Editor* (application 2). The system calls are monitored within the Aurasium framework in a real Android environment. The implemented hook functions have been integrated into the testing application. Thus, it is capable of reading and writing from/to the file. The result of the testing has been recorded by design into the log files.

Also, the content of the *log_taint_map.txt* has been significant, because there is tracked the content of all tainting structures in a time that is the only relevant output of tainting the process. The final restriction is tested performing the black-box approach with the manually repackaged Application 1 and Application 2.

The manual repackaging is necessary in the case of experiments with the ability to perform a step-by-step process of the execution and debugging possibility. When the application is repacked by an automated script, the result of this is installed application package without access to the source code with the following compilation and therefore the debugging process is not available. Concerning application 1, the tainting mechanism has been tested. Final restriction is based on two different methods and communication with the configuration activity. For instance, the tainting mechanism, in this case, performed actions such as *mode, copy, rename, open*, and *send* to test the real hardened application. Operation of removal (*delete)* is not considered as necessary in the manner of the aim of this prototype experiments.

The second of the proposed application 2 is used primarily for testing of specific restriction. There are conducted three scenarios such as opening the unprotected (public) file, opening the protected (private) file in empty data falsifying mode and opening the protected file in a fake data forging type. The primary purpose of these experiments is to focus on data sharing (sending) via various possible channels, such as bluetooth, Wi-Fi, SMS, and email. Note that the sharing is the specific name for sending data to other available applications or other devices connected by open protocols. The method name sharing comes from

the technical aspects of a mobile operating system, in this case, Android. Each sharing method was tested on various types of media files that are usually available on the mobile device. For instance, these file format consists of text files, images, animated images, movie clips, music files, various document files and other types of specific data.

9.1 Sharing Methods

The required behavior of the sharing method is depicted in the table 9.1. There are covered the most useful methods available on the mobile device that is in factory reset state. Related to Android mobile operating system, the factory state of the device contains a selection of applications, which provide the ability to share or communicating via various channels. These applications are selected expressed in this table 9.1 and they are considered as required for implementation experiments.

Sharing method	**Application**	**Outcome without restriction**	**Outcome with restriction**
Email	Gmail	Email is sent	Email is not sent
Bluetooth	Bluetooth application	File is sent (to paired device)	File is not sent (to paired device)
SMS/MMS	Messages	SMS/MMS is sent with attachment	SMS/MMS is not sent
Instant Messaging	Google+	Message with attachment is sent	Message is not sent
Cloud	Google Drive	File is uploaded	File is not uploaded

Table 9.1: Required behavior of sharing method

The interception of the pre-installed application is not considered as satisfactory even the installation package modification was complete without errors. The required behavior has been confirmed in the unrestricted mode, but with the restriction, these application is not even started. From the security perspective, it has the same behavior as the purpose of the thesis require. In contrast, there is no guarantee when the application starts successfully that the behavior is decent as proposed.

To confirm this, it is necessary to randomly select and download another application from the official Google Play market with the same amount of sharing methods. These third-party applications are set as default endpoints of the sharing methods. Applications which are not pre-installed on the device as default are repacked successfully and modified in the same way as the default ones. Also, there are some measure results from the experiments described in the table 9.2.

9.2 Repackaging of Application

As was already discussed, the default applications for sharing data through various channels are failing in the restricted mode. Applications of other vendors are successfully modified and when the user attempts to share data through application 1 via channels described in the table 9.2. A test was performed on the application 1 repackaged using automatic Aurasium script, and the result has the expected isomorphic behavior. The application graphical user interface responds as expected and the private data are protected in all testing cases.

Sharing method	Application	Outcome without restriction	Outcome with restriction
Email	Gmail	Email is sent	Application is not started
	Email.cz	Email is sent	Application is started with empty attachment
Bluetooth	Bluetooth application	File is sent (to paired device)	File sending failed on sending device
SMS/MMS	Messages	SMS/MMS is sent with attachment	Application is not started
	Textra SMS	SMS/MMS is sent with attachment	Application is started without attachment
Instant Messaging	Google+	Message with attachment is sent	Application is not started
	Hangouts+	Message with attachment is sent	Application is not started
	Facebook Messenger	Message with attachment is sent	Application is started without attachment
Cloud	Google Drive	File is uploaded	Application is not started
	Dropbox	File is uploaded	Application is started with empty attachment to upload

Table 9.2: The results of sharing method on third party Applications

This script is part of the Aurasium project, and the overall evaluation of the modification process of application packages is depicted in table 9.3 introduced by [233], in which the row in the table modifies the applications downloaded from the official application market (Google Play). The second row depicts the same applications downloaded from the third-party application market, and 1260 of 3491 were malicious. The results of repackaging script are nearly 100% even the applications are malicious.

Type of Application	Amount of applications	Repackaging success rate
Application store corpus	3491	99.6% (3476)
Malware corpus	1260	99.8% (1258)

Table 9.3: Repackaging evaluation results [233]

Repackaging process introduces the negligible part of the Java code, and the result size of the introduced code in C language is under 50 KB. It is not relevant compared to the vast libraries, and another code is already included in the implementation of Aurasium project. The size of the application package increase as expected. More details about the size overhead and their comparison can be found in [233].

9.3 Performance Overhead

In regards to the performance evaluation, the test has been conducted on real Android device *LG G4* with 1.8 GHz processor (6 cores), 32 GB of internal memory, and 3 GB of system memory (RAM). Start-up time of hardened applications is considerably changed. Aurasium's over-writing of the global offset table entries is time-consuming operation and lasts about 10 s as is shown in the table 9.4.

Application	Original (ms)			Repackaged (ms)		
Application 1	729	830	645	11132	10238	10694
Application 2	1016	1056	1184	11187	11040	11236

Table 9.4: Start-up time overhead on repackaged applications

The performance of the selected applications and actions was tested on the original version of applications (before repackaging) and after the repackaging process with the inactive tainting and restriction and then with the tainting mechanism and current restriction. The target of this evaluation is to identify the application performance from the user perspective and detection of unexpected performance issues. These issues can lead to the discovery of unknown errors and covert vulnerabilities. To determine real performance values the logging mechanism is disabled during measurement. The reason is to have accurate performance values as possible. There is also calculated the overhead of the repackaged application with active functional protection. The performance of restricting was tested on two sharing actions. The first one is the ability to share data through email client and sending away from the device. Bluetooth was the second action with the ability to send any files to a paired (trusted) device. The results of this performance experiments are divided into the sharing methods.

Method	Original [ms]			Inactive [ms]			Active [ms]		
Mediation	1079	1047	922	1469	1258	1515	1320	1368	1336
File protection	1079	1047	922	1469	1258	1515	1226	1297	1351

Table 9.5: Performance of Sharing method through Email

Table 9.5 with the experiments related to the email sharing defines time measurement the performance overhead. Columns describe the states of the application such as original state that is the application without any modifications related to tainting. States named inactive and active means the restriction of the application which can be enabled or disabled, but the application is modified in the way of performing the tainting mechanism through the Aurasium framework. These columns are the same for the second sharing method which is bluetooth and its results are in table 9.6. The overhead in this table is significantly more arduous than in the email sharing method. This overhead is caused by different implementation and reaction on an unexpected condition in the restriction.

Method	Original [ms]			Inactive [ms]			Active [ms]		
Mediation	445	382	508	992	953	914	985	961	1000
File protection	445	382	508	992	953	914	1144	1047	1063

Table 9.6: Performance of Sharing method through Bluetooth

Time from table 9.5 and table 9.6 are read from Android log messages defined for the tainting process. As is shown both tables the time of processing is more related to the application that is responsible for handling the action of the sharing than the method of restriction. The tainting performance was tested for combinations of scanning type and protection of copied file regarding configuration. Moreover, there is measured a duration of a paste action between the start and the end of the copy process. This processes time duration is depicted on table 9.7 and table 9.8. The first table contains the duration of copying for the untainted files and the second for tainted files.

Scanning	Original [ms]			Inactive [ms]			Active [ms]		
File-based	2394	2479	2421	2675	2643	2647	2787	2608	2778
Content-based	2394	2479	2421	2675	2643	2647	2665	2784	2586

Table 9.7: Tainting performance during copying of untainted file

Scanning	Original [ms]			Inactive [ms]			Active [ms]		
File-based	2394	2479	2421	2675	2643	2647	2707	2626	2596
Content-based	2394	2479	2421	2675	2643	2647	2792	2735	2663

Table 9.8: Tainting performance during copying of tainted file

In regards to a user, the results are satisfactory in this case, because the performance of the hardened application remains almost without any time overheads. The table also details the difference between original application and the same application with modification in its inactive and active restrictions state.

The last experiment is focused on the performance of file opening and reading process during increased threat level mode. The worst case results are obtained during the active protection with fake data falsifying because the data needs to be overwritten in the system memory as was described earlier. A faster method is reached with the empty falsifying protection in comparison to the previous method. Unfortunately, there is still about 20% performance overhead against the original unmodified application. Experiment duration times are recorded in the table 9.9.

Data type	Original [ms]			Inactive [ms]			Active [ms]		
Empty data	736	732	704	1881	1425	1357	948	859	816
Fake data	736	732	704	1881	1425	1357	1901	1438	1513

Table 9.9: Duration of file opening during increased threat level

9.4 Summary

To summarize the performance experiments, the start-up time overhead of repackaged applications is the most prominent performance drawback. Otherwise, there can be some slowdowns which are almost transparent to the user and does not represent an obstacle for usage. The average time overhead is in most cases similar in compared to the proposed interception actions in Aurasium framework.

Implementation of the prototype has its limitation. First of all, some applications with activated restriction behave correctly in a secure manner but the not responding state which results in the application failure is not required behavior. To perform user-friendly responds the prototype should be improved and the implementation should be prepared entirely different for each version of the Android operating system.

The main weakness of the solution is the mandatory process of repackaging the application. There is no possibility of tainting the function calls provided by the system itself. Even the development mode of the system does not provide this feature. In that case, the repackaging of the applications is necessary to obtain the control over the function calls or the whole behavior. It is still not complete for the setup the environment with the full control. When these hardened applications are installed, and the configuration is prepared

for the restriction and if the Aurasium framework is missing the management of the application in the meaning of tainting is not working properly. Moreover, if the whole environment is set up as required and the user would like to do something that is not available with the restriction, he can uninstall the Aurasium application or just re-install the application itself with the original one from the official application market store.

This prototype does not cover all situations that can be achieved on the device. For the full protection (even with the application fails during restriction mode) the Aurasium framework and configuration application need to be installed within the system image of the system. The same is achieved by the vendors of the mobile devices. These vendors usually ad a few applications to the original operating system, such as graphical user interface, and third-party applications.

The implementation of the prototype shows the possibility that this approach of dynamically changing the rights of the application is reachable without administration rights, but with limitation to follow specific rule set.

Chapter 10

Conclusion

This thesis analyses the security threats on a mobile device with the focus on privacy protection in the data leakage area. The novel approach of working with sensitive data was presented and defined formally. Moreover, the prototype was introduced, and its model verified through model checking. The high-level goal of this thesis was investigated privacy protection on a mobile device and current solutions provided by the manufacturers of these devices and to find a method of improving the protection of sensitive information.

The contribution of this thesis can be divided into two parts. The first one defines the concept of required behavior to working with public and private files on the same device, presented in chapter 5. The concept consists of restriction mechanism which controls the application system calls and decides if the system call is performed or not related to the opened file. The idea is based on the BYOD principle, which defines the usability of the personal mobile device in the working environment. The concept discussed all related topics such as the design of the required behavior, a framework that can be used and also the implementation of the prototype. The prototype implementation is considered as proof of concept, that was firstly defined by the formal method, described in chapter 6. The prototype was implemented on the open-source platform, which was also identified in the thesis from the security and architecture point of view.

The second contribution of this thesis is verification of presented models to prove that the implementation solution satisfies the required behavior, demonstrated in chapter 7. Model of required behavior and model of implementation were defined and later used for the formal verification process. The formal method was chosen to provide the possibility of portability to other platforms.

The verification was defined in the Uppaal platform, and both models were transformed into the format of this tool. Examples of verification process demonstrate the usability of the proposed method, presented in chapter 8.

Further research can be focused on finding a more specific solution for protecting user data. For instance, artificial intelligence can be considered in this area, at least for the categorization of files into two groups - public and private. Moreover, artificial intelligence can be used to decide which system call can be allowed or denied for a specific file. It is a new era of controlling the content of a mobile device, but the user is considered a person, and there could be very difficult to categorize files and the behavior of user without any knowledge about the user and also the working environment.

The results presented in this thesis were published as a chapter in the book [16], international conferences [17, 18, 19, 100] and in the journals [20] and [218].

Bibliography

[1] Achara, J. P.; Cunche, M.; Roca, V.; et al.: Short paper: WifiLeaks: underestimated privacy implications of the access_wifi_state android permission. In *Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks*. ACM. 2014. pp. 231–236.

[2] Ahmaro, I. Y.; Mustafa, M.; Al-Ahmad, A.: Solaris Operating System. *ALMADINAH ISLAMIC STUDIES*. vol. 1, no. 66. 2012.

[3] Aleksandar Gargenta: Deep Dive into Android IPC/Binder Framework. `https://thenewcircle.com/s/post/1340/Deep_Dive_Into_Binder_Presentation.htm`. 2012.

[4] Alex Lockwood: Binders and Window Tokens. `http://www.androiddesignpatterns.com/2013/07/binders-window-tokens.html`. 2013.

[5] Allbery, B. S.; Bostic, K.; Eckhardt, D.; et al.: Filesystem hierarchy standard. 2015.

[6] Alliance, O. H.: Android open source project. 2011.

[7] Alliance, O. H.: Android (operating system). *Marketing*. vol. 4, no. 5. 2013.

[8] Alur, R.; Dill, D. L.: A theory of timed automata. *Theoretical computer science*. vol. 126, no. 2. 1994: pp. 183–235.

[9] Anderson, J.: *Appcelerator Titanium: Up and Running*. " O'Reilly Media, Inc.„. 2013.

[10] Anderson, J. P.: Computer Security Technology Planning Study. Volume 2. Technical report. Anderson (James P) and Co Fort Washington PA. 1972.

[11] Anderson, R. J.: *Security engineering: a guide to building dependable distributed systems*. John Wiley & Sons. 2010.

[12] Andrus, J.; Nieh, J.: Teaching operating systems using android. In *Proceedings of the 43rd ACM technical symposium on Computer Science Education*. ACM. 2012. pp. 613–618.

[13] Armando, A.; Arsac, W.; Avanesov, T.; et al.: The AVANTSSAR platform for the automated validation of trust and security of service-oriented architectures. *Tools and Algorithms for the Construction and Analysis of Systems*. 2012: pp. 267–282.

[14] Armando, A.; Basin, D.; Boichut, Y.; et al.: The AVISPA tool for the automated validation of internet security protocols and applications. In *International conference on computer aided verification*. Springer. 2005. pp. 281–285.

[15] Armando, A.; Carbone, R.; Compagna, L.: SATMC: A SAT-Based Model Checker for Security-Critical Systems. In *TACAS*, vol. 8413. 2014. pp. 31–45.

[16] Aron, L.: Security Threats on Mobile Devices. In *New Threats and Countermeasures in Digital Crime and Cyber Terrorism*. IGI Global. 2015. pp. 30–52.

[17] Aron, L.; Hanácek, P.: Introduction to Android 5 Security. In *SOFSEM (Student Research Forum Papers/Posters)*. 2015. pp. 103–111.

[18] Aron, L.; Hanacek, P.: Overview of security on mobile devices. In *Web Applications and Networking (WSWAN), 2015 2nd World Symposium on*. IEEE. 2015. pp. 1–11.

[19] Aron, L.; Hanacek, P.: A concept of dynamic permission mechanism on android. In *AIP Conference Proceedings*, vol. 1705. AIP Publishing. 2016. page 020022.

[20] Aron, L.; Hanacek, P.: Dynamic Permission Mechanism on Android. *JSW*. vol. 11, no. 12. 2016: pp. 1124–1230.

[21] Artenstein, N.; Revivo, I.: Man in the binder: He who controls ipc, controls the droid. In *Europe BlackHat Conf.*. 2014.

[22] Arzt, S.; Rasthofer, S.; Bodden, E.: Susi: A tool for the fully automated classification and categorization of android sources and sinks. 2013.

[23] Arzt, S.; Rasthofer, S.; Fritz, C.; et al.: Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices*. vol. 49, no. 6. 2014: pp. 259–269.

[24] Avdiienko, V.; Kuznetsov, K.; Gorla, A.; et al.: Mining apps for abnormal usage of sensitive data. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press. 2015. pp. 426–436.

[25] Babil, G. S.; Mehani, O.; Boreli, R.; et al.: On the effectiveness of dynamic taint analysis for protecting against private information leaks on Android-based devices. In *Security and Cryptography (SECRYPT), 2013 International Conference on*. IEEE. 2013. pp. 1–8.

[26] Bach, M. J.; et al.: *The design of the UNIX operating system*. vol. 1. Prentice-Hall Englewood Cliffs, NJ. 1986.

[27] Backes, M.; Bugiel, S.; Gerling, S.; et al.: Android Security Framework: Extensible multi-layered access control on Android. In *Proceedings of the 30th annual computer security applications conference*. ACM. 2014. pp. 46–55.

[28] Baclit, R.; Sicam, C.; Membrey, P.; et al.: Bash. *Foundations of CentOS Linux*. 2009: pp. 31–54.

[29] Ballagas, R.; Rohs, M.; Sheridan, J. G.; et al.: Byod: Bring your own device. In *Proceedings of the Workshop on Ubiquitous Display Environments, Ubicomp*, vol. 2004. 2004.

[30] Banuri, H.; Alam, M.; Khan, S.; et al.: An Android runtime security policy enforcement framework. *Personal and Ubiquitous Computing*. vol. 16, no. 6. 2012: pp. 631–641.

[31] Bartholomew, D.: Qemu a multihost multitarget emulator. *Linux Journal*. vol. 2006, no. 145. 2006: page 3.

[32] Basin, D.; Mödersheim, S.; Vigano, L.: OFMC: A symbolic model checker for security protocols. *International Journal of Information Security*. vol. 4, no. 3. 2005: pp. 181–208.

[33] Baumann, A.; Heiser, G.; Appavoo, J.; et al.: Providing Dynamic Update in an Operating System. In *USENIX Annual Technical Conference, General Track*. 2005. pp. 279–291.

[34] Bell, D. E.; LaPadula, L. J.: Secure computer systems: Mathematical foundations. Technical report. MITRE CORP BEDFORD MA. 1973.

[35] Benantar, M.: *Access control systems: security, identity management and trust models*. Springer Science & Business Media. 2006.

[36] Bengtsson, J.; Larsen, K.; Larsson, F.; et al.: UPPAAL—a tool suite for automatic verification of real-time systems. *Hybrid Systems III*. 1996: pp. 232–243.

[37] Bettany, A.; Halsey, M.: What Is Malware? In *Windows Virus and Malware Troubleshooting*. Springer. 2017. pp. 1–8.

[38] Bhagwat, P.: Bluetooth: technology for short-range wireless apps. *IEEE Internet Computing*. vol. 5, no. 3. 2001: pp. 96–103.

[39] Bhavani, A.: Cross-site scripting attacks on android webview. *arXiv preprint arXiv:1304.7451*. 2013.

[40] Biba, K. J.: Integrity considerations for secure computer systems. Technical report. MITRE CORP BEDFORD MA. 1977.

[41] Birman, K. P.: Remote Procedure Calls and the Client/Server Model. In *Guide to Reliable Distributed Systems*. Springer. 2012. pp. 185–247.

[42] Bishop, M. A.: *The art and science of computer security*. Addison-Wesley Longman Publishing Co., Inc.. 2002.

[43] Bjørner, N.; Browne, A.; Chang, E.; et al.: STeP: Deductive-algorithmic verification of reactive and real-time systems. In *International Conference on Computer Aided Verification*. Springer. 1996. pp. 415–418.

[44] Bodden, E.; Hermann, B.; Lerch, J.; et al.: Reducing human factors in software security architectures. In *Future Security Conference (to appear*. 2013.

[45] Bodei, C.; Buchholtz, M.; Degano, P.; et al.: Automatic validation of protocol narration. In *Computer Security Foundations Workshop, 2003. Proceedings. 16th IEEE*. IEEE. 2003. pp. 126–140.

[46] Brockmann, A.: A Plausibly Deniable Encryption Scheme for Personal Data Storage. 2015.

[47] Brookes, S. T.; Whitley, E.; Peters, T. J.; et al.: Subgroup analyses in randomised controlled trials: quantifying the risks of false-positives and false-negatives. *Health Technology Assessment*. vol. 5, no. 33. 2001: pp. 1–56.

[48] Bugnion, E.; Devine, S.; Rosenblum, M.; et al.: Bringing virtualization to the x86 architecture with the original vmware workstation. *ACM Transactions on Computer Systems (TOCS)*. vol. 30, no. 4. 2012: page 12.

[49] Burnette, E.: *Eclipse IDE Pocket Guide*. O'Reilly Media, Inc.. 2005.

[50] Cabrero, J. E.; Holland, I. M.: System and method for providing shared global offset table for common shared library in a computer system. July 10 2001. uS Patent 6,260,075.

[51] Chang, B.; Wang, Z.; Chen, B.; et al.: Mobipluto: File system friendly deniable storage for mobile devices. In *Proceedings of the 31st Annual Computer Security Applications Conference*. ACM. 2015. pp. 381–390.

[52] Chen, H.; Wagner, D.; Dean, D.: Setuid Demystified. In *USENIX Security Symposium*. 2002. pp. 171–190.

[53] Chiang, H.-Y.; Chiasson, S.: Improving user authentication on mobile devices: A touchscreen graphical password. In *Proceedings of the 15th international conference on Human-computer interaction with mobile devices and services*. ACM. 2013. pp. 251–260.

[54] Chin, E.; Felt, A. P.; Greenwood, K.; et al.: Analyzing inter-application communication in Android. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*. ACM. 2011. pp. 239–252.

[55] Chin, E.; Wagner, D.: Bifocals: Analyzing webview vulnerabilities in android applications. In *International Workshop on Information Security Applications*. Springer. 2013. pp. 138–159.

[56] Cinar, O.: Android Platform. In *Android Quick APIs Reference*. Springer. 2015. pp. 1–14.

[57] Clarke, E. M.; Grumberg, O.; Peled, D.: *Model checking*. MIT press. 1999.

[58] Cremers, C. J.: The scyther tool: Verification, falsification, and analysis of security protocols. In *CAV*, vol. 8. Springer. 2008. pp. 414–418.

[59] Cunche, M.: I know your MAC Address: Targeted tracking of individual using Wi-Fi. *Journal of Computer Virology and Hacking Techniques*. vol. 10, no. 4. 2014: pp. 219–227.

[60] Dabbish, L.; Stuart, C.; Tsay, J.; et al.: Social coding in GitHub: transparency and collaboration in an open software repository. In *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work*. ACM. 2012. pp. 1277–1286.

[61] Danford, T. E.; Batchu, S. K.: Virtual instance architecture for mobile device management systems. November 15 2011. uS Patent 8,060,074.

[62] Davi, L.; Dmitrienko, A.; Sadeghi, A.-R.; et al.: Privilege escalation attacks on android. In *International Conference on Information Security*. Springer. 2010. pp. 346–360.

[63] Deering, S. E.: Internet protocol, version 6 (IPv6) specification. 1998.

[64] Denning, D. E.: A lattice model of secure information flow. *Communications of the ACM*. vol. 19, no. 5. 1976: pp. 236–243.

[65] Desmedt, Y.: Man-in-the-middle attack. In *Encyclopedia of cryptography and security*. Springer. 2011. pp. 759–759.

[66] Developers, A.: Adt plugin for eclipse. *URI: http://devel-oper. android. com/sdk/eclipse-adt. html*. 2012.

[67] Dey, S.; Roy, N.; Xu, W.; et al.: AccelPrint: Imperfections of Accelerometers Make Smartphones Trackable. In *NDSS*. 2014.

[68] Drake, J. J.; Lanier, Z.; Mulliner, C.; et al.: *Android Hacker's Handbook*. John Wiley & Sons. 2014.

[69] Dumas, M.; Ter Hofstede, A. H.: UML activity diagrams as a workflow specification language. In *UML*, vol. 2185. Springer. 2001. pp. 76–90.

[70] Ehringer, D.: The dalvik virtual machine architecture. *Techn. report (March 2010)*. vol. 4. 2010: page 8.

[71] Elenkov, N.: *Android security internals: An in-depth guide to Android's security architecture*. No Starch Press. 2014.

[72] 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)*. vol. 32, no. 2. 2014: page 5.

[73] Enck, W.; Octeau, D.; McDaniel, P.; et al.: A Study of Android Application Security. In *USENIX security symposium*, vol. 2. 2011. page 2.

[74] Enck, W.; Ongtang, M.; McDaniel, P. D.; et al.: Understanding Android Security. *IEEE security & privacy*. vol. 7, no. 1. 2009: pp. 50–57.

[75] Fábrega, F. J. T.; Javier, F.; Guttman, J. D.: Copy on write. 1995.

[76] Fagin, R.: On an authorization mechanism. *ACM Transactions on Database Systems (TODS)*. vol. 3, no. 3. 1978: pp. 310–319.

[77] Felt, A. P.; Finifter, M.; Chin, E.; et al.: A survey of mobile malware in the wild. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*. ACM. 2011. pp. 3–14.

[78] Ferraiolo, D. F.; Sandhu, R.; Gavrila, S.; et al.: Proposed NIST standard for role-based access control. *ACM Transactions on Information and System Security (TISSEC)*. vol. 4, no. 3. 2001: pp. 224–274.

[79] Flanagan, D.: *JavaScript: the definitive guide*. “ O'Reilly Media, Inc.„. 2006.

[80] Forman, I. R.; Forman, N.; Ibm, J. V.: Java reflection in action. 2004.

[81] Forouzan, B. A.: *TCP/IP protocol suite*. McGraw-Hill, Inc.. 2002.

[82] Fritz, C.; Arzt, S.; Rasthofer, S.; et al.: Highly precise taint analysis for Android applications. EC SPRIDE. Technical report. TU Darmstadt, Tech. Rep. 2013.

[83] Furnell, S.; Clarke, N.; Karatzouni, S.: Beyond the pin: Enhancing user authentication for mobile devices. *Computer fraud & security*. vol. 2008, no. 8. 2008: pp. 12–17.

[84] Gabbay, D. M.; Hodkinson, I.; Reynolds, M.: Temporal Logic Mathematical Foundations and Computational Aspects. 1994.

[85] Garcia, J.; De Moss, A.; Simoens, M.: *Sencha Touch in action*. Manning Publications. 2013.

[86] Gargenta, A.: Deep dive into Android IPC/Binder framework. *Android Builders Summit*. 2013.

[87] Garrity, D. F.: Binding apparatus. May 16 2000. uS Patent 6,062,792.

[88] Gelter, A.; Parker, B.; Boatright, R.; et al.: Memory management unit. 2013. uS Patent 8,443,098.

[89] Gibler, C.; Crussell, J.; Erickson, J.; et al.: AndroidLeaks: automatically detecting potential privacy leaks in android applications on a large scale. In *International Conference on Trust and Trustworthy Computing*. Springer. 2012. pp. 291–307.

[90] Gilmore, S.; Haenel, V.; Kloul, L.; et al.: Choreographing security and performance analysis for web services. *Formal Techniques for Computer Systems and Business Processes*. 2005: pp. 200–214.

[91] Gordon, R.: *Essential JNI: Java Native Interface*. Prentice-Hall, Inc.. 1998.

[92] Gosling, J.: *The Java language specification*. Addison-Wesley Professional. 2000.

[93] Götzfried, J.; Müller, T.: Analysing Android's Full Disk Encryption Feature. *JoWUA*. vol. 5, no. 1. 2014: pp. 84–100.

[94] Gu, G.; Porras, P. A.; Yegneswaran, V.; et al.: BotHunter: Detecting Malware Infection Through IDS-Driven Dialog Correlation. In *Usenix Security*, vol. 7. 2007. pp. 1–16.

[95] Guellati, S.; Kitouni, I.; Saidouni, D.-e.: Verification of durational action timed automata using uppaal. *International Journal of Computer Applications*. vol. 56, no. 11. 2012.

[96] Haack, C.; Jeffrey, A.: Timed spi-calculus with types for secrecy and authenticity. In *CONCUR*, vol. 5. Springer. 2005. pp. 202–216.

[97] Hafer, T.; Thomas, W.: Computation tree logic CTL* and path quantifiers in the monadic theory of the binary tree. *Automata, Languages and Programming*. 1987: pp. 269–279.

[98] Halpert, B.: Mobile device security. In *Proceedings of the 1st annual conference on Information security curriculum development*. ACM. 2004. pp. 99–101.

[99] Han, Y.; Chronopoulos, A. T.: Distributed loop scheduling schemes for cloud systems. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*. IEEE. 2013. pp. 955–962.

[100] Hanacek, L. A. P.: Mobile Security for Banking on Android Platform. In *The International Conference on Computing Technology, Information Security and Risk Management (CTISRM2016)*. 2016. page 7.

[101] Harrison, M. A.; Ruzzo, W. L.; Ullman, J. D.: Protection in operating systems. *Communications of the ACM*. vol. 19, no. 8. 1976: pp. 461–471.

[102] Haselsteiner, E.; Breitfuß, K.: Security in near field communication (NFC). In *Workshop on RFID security*. 2006. pp. 12–14.

[103] Hatcher, E.; Loughran, S.: *Java development with Ant*. Manning: London: Pearson Education,. 2003.

[104] Herken, R.: The Universal Turing Machine. A Half-Century Survey. 1992.

[105] Heuser, S.; Nadkarni, A.; Enck, W.; et al.: ASM: A Programmable Interface for Extending Android Security. In *USENIX Security*, vol. 14. 2014. pp. 1005–1109.

[106] Hirviniemi, S.: Wide area network (wan) interface for a transmission control protocol/internet protocol (tcp/ip) in a local area network (lan). September 1 1998. uS Patent 5,802,285.

[107] Holla, S.; Katti, M. M.: Android based mobile application development and its security. *International Journal of Computer Trends and Technology*. vol. 3, no. 3. 2012: pp. 486–490.

[108] Holzmann, G.: *Spin model checker, the: primer and reference manual*. Addison-Wesley Professional. 2003.

[109] Hoog, A.: *Android forensics: investigation, analysis and mobile security for Google Android*. Elsevier. 2011.

[110] Hoog, A.; Strzempka, K.: *iPhone and iOS forensics: Investigation, analysis and mobile security for Apple iPhone, iPad and iOS devices*. Elsevier. 2011.

[111] 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.

[112] Hu, C.; Neamtiu, I.: Automating GUI testing for Android applications. In *Proceedings of the 6th International Workshop on Automation of Software Test*. ACM. 2011. pp. 77–83.

[113] Huang, J.: Android IPC Mechanism. *Southern Taiwan University of Technology*. 2012.

[114] Jann, J.; Dubey, N.; Burugula, R. S.; et al.: Dynamic reconfiguration of CPU and WebSphere on IBM pSeries servers. *Software: Practice and Experience*. vol. 34, no. 13. 2004: pp. 1257–1272.

[115] Jensen, H. E.; Larsen, K. G.; Skou, A.: Modelling and Analysis of a Collision Avoidance Protocol using SPIN and UPPAAL. *BRICS Report Series*. vol. 3, no. 24. 1996.

[116] Jing, Y.; Ahn, G.-J.; Zhao, Z.; et al.: Riskmon: Continuous and automated risk assessment of mobile applications. In *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*. ACM. 2014. pp. 99–110.

[117] Johnson, M. K.; Troan, E. W.: *Linux application development*. Addison-Wesley Professional. 2004.

[118] Kajiwara, N.; Matsumoto, S.; Nishimoto, Y.; et al.: Detection of Privacy Sensitive Information Retrieval Using API Call Logging Mechanism within Android Framework. *JNW*. vol. 9, no. 11. 2014: pp. 2905–2913.

[119] Kaladharan, Y.; Mateti, P.; Jevitha, K.: An Encryption Technique to Thwart Android Binder Exploits. In *Intelligent Systems Technologies and Applications*. Springer. 2016. pp. 13–21.

[120] Karlsson, K.-J.; Glisson, W. B.: Android Anti-forensics: Modifying CyanogenMod. In *System Sciences (HICSS), 2014 47th Hawaii International Conference on*. IEEE. 2014. pp. 4828–4837.

[121] Kernighan, B. W.; Ritchie, D. M.: *The C programming language*. 2006.

[122] Kim, J.; Yoon, Y.; Yi, K.; et al.: ScanDal: Static analyzer for detecting privacy leaks in android applications. *MoST*. vol. 12. 2012.

[123] Kim, Y.-J.; Cho, S.-J.; Kim, K.-J.; et al.: Benchmarking Java application using JNI and native C application on Android. In *Control, Automation and Systems (ICCAS), 2012 12th International Conference on*. IEEE. 2012. pp. 284–288.

[124] Kivity, A.; Kamay, Y.; Laor, D.; et al.: kvm: the Linux virtual machine monitor. In *Proceedings of the Linux symposium*, vol. 1. 2007. pp. 225–230.

[125] Kortesalmi, J.; Pelto, T.: Preventing misuse of a copied subscriber identity in a mobile communication system. July 30 2002. uS Patent 6,427,073.

[126] Kosoresow, A. P.; Hofmeyer, S.: Intrusion detection via system call traces. *IEEE software*. vol. 14, no. 5. 1997: pp. 35–42.

[127] Kostakos, V.; O'Neill, E.: NFC on mobile phones: issues, lessons and future research. In *Pervasive Computing and Communications Workshops, 2007. PerCom Workshops' 07. Fifth Annual IEEE International Conference on*. IEEE. 2007. pp. 367–370.

[128] Krishnamurthy, B.; Wills, C. E.: Privacy leakage in mobile online social networks. In *Proceedings of the 3rd Wonference on Online social networks*. USENIX Association. 2010. pp. 4–4.

[129] La Polla, M.; Martinelli, F.; Sgandurra, D.: A survey on security for mobile devices. *IEEE communications surveys & tutorials*. vol. 15, no. 1. 2013: pp. 446–471.

[130] Lamport, L.: The temporal logic of actions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*. vol. 16, no. 3. 1994: pp. 872–923.

[131] Lampson, B. W.: Protection. *ACM SIGOPS Operating Systems Review*. vol. 8, no. 1. 1974: pp. 18–24.

[132] Lange, M.; Liebergeld, S.: Crossover: secure and usable user interface for mobile devices with multiple isolated os personalities. In *Proceedings of the 29th Annual Computer Security Applications Conference*. ACM. 2013. pp. 249–257.

[133] Lange, M.; Liebergeld, S.; Lackorzynski, A.; et al.: L4Android: a generic operating system framework for secure smartphones. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*. ACM. 2011. pp. 39–50.

[134] Larsen, K. G.; Pettersson, P.; Yi, W.: Model-checking for real-time systems. In *International Symposium on Fundamentals of Computation Theory*. Springer. 1995. pp. 62–88.

[135] Lee, H.; Chuvyrov, E.; Ferracchiati, F. C.: *Beginning Windows Phone 7 Development*. Springer. 2010.

[136] Lee, I.; Sokolsky, O.: A graphical property specification language. In *High-Assurance Systems Engineering Workshop, 1997., Proceedings*. IEEE. 1997. pp. 42–47.

[137] Lewine, D.: *POSIX programmers guide*. " O'Reilly Media, Inc.„. 1991.

[138] Li, L.; Zhao, X.; Xue, G.: Unobservable Re-authentication for Smartphones. In *NDSS*. 2013.

[139] Li, N.: Discretionary access control. In *Encyclopedia of Cryptography and Security*. Springer. 2011. pp. 353–356.

[140] Liang, H.; Wu, D.; Xu, J.; et al.: Survey on privacy protection of android devices. In *Cyber Security and Cloud Computing (CSCloud), 2015 IEEE 2nd International Conference on*. IEEE. 2015. pp. 241–246.

[141] License, G. G. P.: GNU General Public License. *Retrieved December*. vol. 25. 1989: page 2014.

[142] Liebergeld, S.; Lange, M.: Android security, pitfalls and lessons learned. In *Information Sciences and Systems 2013*. Springer. 2013. pp. 409–417.

[143] Lindahl, M.; Pettersson, P.; Yi, W.: Formal design and analysis of a gear controller: An industrial case study using uppaal. In *LNCS, Proc. of the 4th International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, vol. 1384. 1998. pp. 281–297.

[144] Lindqvist, H.: Mandatory access control. *Master's Thesis in Computing Science, Umea University, Department of Computing Science, SE-901*. vol. 87. 2006.

[145] Liu, C. Z.; Au, Y. A.; Choi, H. S.: Effects of freemium strategy in the mobile app market: an empirical study of Google play. *Journal of Management Information Systems*. vol. 31, no. 3. 2014: pp. 326–354.

[146] Lokhande, B.; Dhavale, S.: Overview of information flow tracking techniques based on taint analysis for android. In *Computing for Sustainable Global Development (INDIACom), 2014 International Conference on*. IEEE. 2014. pp. 749–753.

[147] Love, R.: *Linux Kernel Development (Novell Press)*. Novell Press. 2005.

[148] Lu, K.; Li, Z.; Kemerlis, V. P.; et al.: Checking More and Alerting Less: Detecting Privacy Leakages via Enhanced Data-flow Analysis and Peer Voting. In *NDSS*. 2015.

[149] Maggi, F.; Gasparini, S.; Boracchi, G.: A fast eavesdropping attack against touchscreens. In *Information Assurance and Security (IAS), 2011 7th International Conference on*. IEEE. 2011. pp. 320–325.

[150] Mahesh, B. R.; Kumar, M. B.; Manoharan, R.; et al.: Portability of mobile applications using phonegap: A case study. In *Software Engineering and Mobile Application Modelling and Development (ICSEMA 2012), International Conference on*. IET. 2012. pp. 1–6.

[151] Mainetti, L.; Patrono, L.; Vergallo, R.: IDA-Pay: an innovative micro-payment system based on NFC technology for Android mobile devices. In *Software, Telecommunications and Computer Networks (SoftCOM), 2012 20th International Conference on*. IEEE. 2012. pp. 1–6.

[152] Makino, H.; Ishii, I.; Nakashizuka, M.: Development of navigation system for the blind using GPS and mobile phone combination. In *Engineering in Medicine and Biology Society, 1996. Bridging Disciplines for Biomedicine. Proceedings of the 18th Annual International Conference of the IEEE*, vol. 2. IEEE. 1996. pp. 506–507.

[153] Marforio, C.; Ritzdorf, H.; Francillon, A.; et al.: Analysis of the communication between colluding applications on modern smartphones. In *Proceedings of the 28th Annual Computer Security Applications Conference*. ACM. 2012. pp. 51–60.

[154] Mauerer, W.: *Professional Linux kernel architecture*. John Wiley & Sons. 2010.

[155] McKerley, M.; Lohner, M.; Fulay, A.: Session S316970: Enabling Database-as-a-Service Through Agile, Self-Service Driven Provisioning. 2010.

[156] Meadows, C. A.: Formal verification of cryptographic protocols: A survey. In *International Conference on the Theory and Application of Cryptology*. Springer. 1994. pp. 133–150.

[157] Meier, R.: *Professional Android 4 application development*. John Wiley & Sons. 2012.

[158] Menezes, A. J.; Van Oorschot, P. C.; Vanstone, S. A.: *Handbook of applied cryptography*. CRC press. 1996.

[159] Meyer, J.; Downing, T.: Jasmin. `http://jasmin.sourceforge.net/`. 2017.

[160] Midi, D.; Oluwatimi, O.; Shebaro, B.; et al.: Demo overview: privacy-enhancing features of identidroid. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2014. pp. 1481–1483.

[161] Milette, G.; Stroud, A.: *Professional Android sensor programming*. John Wiley & Sons. 2012.

[162] Miller, K.; Pegah, M.: Virtualization: virtually at the desktop. In *Proceedings of the 35th annual ACM SIGUCCS fall conference*. ACM. 2007. pp. 255–260.

[163] Miller, K. W.; Voas, J.; Hurlburt, G. F.: BYOD: security and privacy considerations. *It Professional*. vol. 14, no. 5. 2012: pp. 0053–55.

[164] Mödersheim, S.; Vigano, L.; Basin, D.: Constraint differentiation: Search-space reduction for the constraint-based analysis of security protocols. *Journal of Computer Security*. vol. 18, no. 4. 2010: pp. 575–618.

[165] Morris, J.; Smalley, S.; Kroah-Hartman, G.: Linux security modules: General security support for the linux kernel. In *USENIX Security Symposium*. 2002.

[166] Musciano, C.; Kennedy, B.; et al.: *HTML, the definitive Guide*. O'Reilly & Associates. 1996.

[167] Nakao, K.; Nakamoto, Y.: Toward remote service invocation in android. In *Ubiquitous Intelligence & Computing and 9th International Conference on Autonomic & Trusted Computing (UIC/ATC), 2012 9th International Conference on*. IEEE. 2012. pp. 612–617.

[168] Oh, H.-S.; Kim, B.-J.; Choi, H.-K.; et al.: Evaluation of Android Dalvik virtual machine. In *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems*. ACM. 2012. pp. 115–124.

[169] OpenIntents: OI File Manager. `https://play.google.com/store/apps/details?id=org.openintents.filemanager`.

[170] Oracle, V.: VirtualBox, User Manual, 2011. 2012.

[171] Papadimitriou, P.; Garcia-Molina, H.: Data leakage detection. *IEEE Transactions on knowledge and data engineering*. vol. 23, no. 1. 2011: pp. 51–63.

[172] Parr, T.: *The definitive ANTLR 4 reference*. Pragmatic Bookshelf. 2013.

[173] Paul, K.; Kundu, T. K.: Android on mobile devices: An energy perspective. In *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*. IEEE. 2010. pp. 2421–2426.

[174] Peng, H.; Gates, C.; Sarma, B.; et al.: Using probabilistic generative models for ranking risks of android apps. In *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM. 2012. pp. 241–252.

[175] Peterson, J. L.; Silberschatz, A.: *Operating system concepts*. vol. 2. Addison-Wesley Reading, MA. 1985.

[176] Pfleeger, C. P.; Pfleeger, S. L.: *Security in computing*. Prentice Hall Professional Technical Reference. 2002.

[177] Pitts, D.; Ball, B.; et al.: *Red Hat Linux*. Sams. 1998.

[178] Qian, C.; Luo, X.; Shao, Y.; et al.: On tracking information flows through jni in android applications. In *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*. IEEE. 2014. pp. 180–191.

[179] Racine, J.: The cygwin tools: a GNU toolkit for windows. 2000.

[180] Ratabouil, S.: *Android NDK: Beginner's Guide*. Packt Publishing Ltd. 2015.

[181] Reps, T.; Horwitz, S.; Sagiv, M.: Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM. 1995. pp. 49–61.

[182] Reshetova, E.; Karhunen, J.; Nyman, T.; et al.: Security of OS-level virtualization technologies: Technical report. *arXiv preprint arXiv:1407.4245*. 2014.

[183] Rhee, K.; Jeon, W.; Won, D.: Security requirements of a mobile device management system. *International Journal of Security and Its Applications*. vol. 6, no. 2. 2012: pp. 353–358.

[184] Rish, I.: An empirical study of the naive Bayes classifier. In *IJCAI 2001 workshop on empirical methods in artificial intelligence*, vol. 3. IBM New York. 2001. pp. 41–46.

[185] Rushby, J.; et al.: A trusted computing base for embedded systems. In *Proceedings 7th DoD/NBS Computer Security Conference*. sn. 1984. pp. 294–311.

[186] Russello, G.; Jimenez, A. B.; Naderi, H.; et al.: Firedroid: Hardening security in almost-stock android. In *Proceedings of the 29th Annual Computer Security Applications Conference*. ACM. 2013. pp. 319–328.

[187] Russinovich, M. E.; Solomon, D. A.; Allchin, J.: *Microsoft Windows Internals: Microsoft Windows Server 2003, Windows XP, and Windows 2000*. vol. 4. Microsoft Press Redmond. 2005.

[188] Salehi, M.; Daryabar, F.; Tadayon, M. H.: Welcome to Binder: A kernel level attack model for the Binder in Android operating system. In *Telecommunications (IST), 2016 8th International Symposium on*. IEEE. 2016. pp. 156–161.

[189] Salinas, A. B.; Esteban, M. C.; Herber, T.: Method and apparatus for communicating data packets from an external packet network to a mobile radio station. October 22 2002. uS Patent 6,469,998.

[190] Samarati, P.; Di Vimercati, S. D. C.: Access control: Policies, models, and mechanisms. *Lecture notes in computer science.* , no. 2171. 2001: pp. 137–196.

[191] Samarati, P.; de Vimercati, S. C.: Access control: Policies, models, and mechanisms. In *International School on Foundations of Security Analysis and Design*. Springer. 2000. pp. 137–196.

[192] Sandhu, R. S.: Role-based access control. 1997.

[193] Sandhu, R. S.; Samarati, P.: Access control: principle and practice. *Communications Magazine, IEEE*. vol. 32, no. 9. 1994: pp. 40–48.

[194] Schreckling, D.; Köstler, J.; Schaff, M.: Kynoid: real-time enforcement of fine-grained, user-defined, and data-centric security policies for android. *information security technical report*. vol. 17, no. 3. 2013: pp. 71–80.

[195] Schreiber, T.: Android binder. *A shorter, more general work, but good for an overview of Binder. http://www. nds. rub. de/media/attachments/files/2012/03/binder. pdf*. 2011.

[196] Schwartz, E. J.; Avgerinos, T.; Brumley, D.: All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Security and privacy (SP), 2010 IEEE symposium on*. IEEE. 2010. pp. 317–331.

[197] Schwarz, B.; Debray, S.; Andrews, G.: Disassembly of executable code revisited. In *Reverse engineering, 2002. Proceedings. Ninth working conference on*. IEEE. 2002. pp. 45–54.

[198] Shabtai, A.; Fledel, Y.; Kanonov, U.; et al.: Google android: A comprehensive security assessment. *IEEE Security & Privacy*. vol. 8, no. 2. 2010: pp. 35–44.

[199] Shalabi, S. M.; Doll, C. L.; Reilly, J. D.; et al.: Access control list. December 5 2011. uS Patent App. 13/311,278.

[200] Shanker, A.; Lai, S.: Android porting concepts. In *Electronics Computer Technology (ICECT), 2011 3rd International Conference on*, vol. 5. IEEE. 2011. pp. 129–133.

[201] Shen, F.; Vishnubhotla, N.; Todarka, C.; et al.: Information flows as a permission mechanism. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM. 2014. pp. 515–526.

[202] Shin, W.; Kiyomoto, S.; Fukushima, K.; et al.: A formal model to analyze the permission authorization and enforcement in the android framework. In *Social Computing (SocialCom), 2010 IEEE Second International Conference on*. IEEE. 2010. pp. 944–951.

[203] Sickert, S.: Linear Temporal Logic. *Archive of Formal Proofs*. vol. 2016. 2016.

[204] Smalley, S.; Craig, R.: Security Enhanced (SE) Android: Bringing Flexible MAC to Android. In *NDSS*, vol. 310. 2013. pp. 20–38.

[205] Smalley, S.; R2X, T. M.: The case for SE Android. *National Security Agency (NSA)*. 2011.

[206] Spivey, J. M.; Abrial, J.: *The Z notation*. Prentice Hall Hemel Hempstead. 1992.

[207] Spride, E: DroidBench – Benchmarks. `http://sseblog.ec-spride.de/tools/droidbench/`. 2017.

[208] Stallman, R. M.; McGrath, R.: *GNU Make: A Program for Directed Recompilation, Version 3. 79. 1.* Free Software Foundation. 2002.

[209] Stevens, W. R.; Fenner, B.; Rudoff, A. M.: *UNIX network programming.* vol. 1. Addison-Wesley Professional. 2004.

[210] Stroustrup, B.: *The C++ programming language.* Pearson Education India. 1995.

[211] Sun, M.; Zheng, M.; Lui, J.; et al.: Design and implementation of an android host-based intrusion prevention system. In *Proceedings of the 30th Annual Computer Security Applications Conference.* ACM. 2014. pp. 226–235.

[212] Takahashi, T.; Nakao, K.; Kanaoka, A.: Data model for android package information and its application to risk analysis system. In *Proceedings of the 2014 ACM Workshop on Information Sharing & Collaborative Security.* ACM. 2014. pp. 71–80.

[213] Tan, D. J.; Chua, T.-W.; Thing, V. L.; et al.: Securing android: a survey, taxonomy, and challenges. *ACM Computing Surveys (CSUR).* vol. 47, no. 4. 2015: page 58.

[214] Tolone, W.; Ahn, G.-J.; Pai, T.; et al.: Access control in collaborative systems. *ACM Computing Surveys (CSUR).* vol. 37, no. 1. 2005: pp. 29–41.

[215] Tripp, O.; Rubin, J.: A Bayesian Approach to Privacy Enforcement in Smartphones. In *USENIX Security*, vol. 14. 2014. pp. 175–190.

[216] Turuani, M.: The CL-Atse protocol analyser. In *International Conference on Rewriting Techniques and Applications.* Springer. 2006. pp. 277–286.

[217] Van Rossum, G.; Drake, F. L.: *The python language reference manual.* Network Theory Ltd.. 2011.

[218] VANČO, B. M.; Aron, L.: Dynamic Security Policy Enforcement on Android. *International Journal of Security and Its Applications.* vol. 10, no. 9. 2016: pp. 141–148.

[219] Velte, A.; Velte, T.: *Microsoft virtualization with Hyper-V.* McGraw-Hill, Inc.. 2009.

[220] Velte, A. T.; Velte, T. J.; Elsenpeter, R. C.; et al.: *Cloud computing: a practical approach.* McGraw-Hill New York. 2010.

[221] Vidas, T.; Votipka, D.; Christin, N.: All Your Droid Are Belong to Us: A Survey of Current Android Attacks. In *WOOT.* 2011. pp. 81–90.

[222] Vise, D.: The google story. *Strategic Direction.* vol. 23, no. 10. 2007.

[223] Von Oheimb, D.; Mödersheim, S.: ASLan++—a formal security specification language for distributed systems. In *Formal Methods for Components and Objects.* Springer. 2011. pp. 1–22.

[224] Waldspurger, C. A.: Memory resource management in VMware ESX server. *ACM SIGOPS Operating Systems Review.* vol. 36, no. SI. 2002: pp. 181–194.

[225] Wang, X.; Sun, K.; Wang, Y.; et al.: DeepDroid: Dynamically Enforcing Enterprise Policy on Android Devices. In *NDSS.* 2015.

[226] Wells, G.: The Future of iOS Development: Evaluating the Swift Programming Language. 2015.

[227] Wu, C.; Zhou, Y.; Patel, K.; et al.: AirBag: Boosting Smartphone Resistance to Malware Infection. In *NDSS*. 2014.

[228] WWW Pages: System Permissions. `http://developer.android.com/guide/topics/security/permissions.html`.

[229] WWW Pages: 99.6 percent of new smartphones run Android or iOS. `https://www.theverge.com/2017/2/16/14634656/android-ios-market-share-blackberry-2016`. 2017.

[230] WWW Pages: Android Security Overview. `https://source.android.com/security/`. 2017.

[231] WWW Pages: Apache Harmony Project. `http://harmony.apache.org/`. 2017.

[232] WWW Pages: Dalvik Executable format. `https://source.android.com/devices/tech/dalvik/dex-format`. 2017.

[233] Xu, R.; Saïdi, H.; Anderson, R. J.: Aurasium: practical policy enforcement for android applications. In *USENIX Security Symposium*, vol. 2012. 2012.

[234] Yaghmour, K.: *Embedded Android: Porting, Extending, and Customizing.* " O'Reilly Media, Inc.„. 2013.

[235] Yan, L.-K.; Yin, H.: DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis. In *USENIX security symposium*. 2012. pp. 569–584.

[236] Yang, S.; Yan, D.; Wu, H.; et al.: Static control-flow analysis of user-driven callbacks in Android applications. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, vol. 1. IEEE. 2015. pp. 89–99.

[237] Yang, X.; Sang, N.; Alves-Foss, J.: Shortening the Boot Time of Android OS. *Computer*. vol. 47, no. 7. 2014: pp. 53–58.

[238] Yang, Z.; Yang, M.: Leakminer: Detect information leakage on android with static taint analysis. In *Software Engineering (WCSE), 2012 Third World Congress on*. IEEE. 2012. pp. 101–104.

[239] Yi, W.; Pettersson, P.; Daniels, M.: Automatic verification of real-time communicating systems by constraint-solving. In *Formal Description Techniques VII*. Springer. 1995. pp. 243–258.

[240] Youngdale, E.: Kernel korner: The ELF object file format by dissection. *Linux Journal*. vol. 1995, no. 13es. 1995: page 15.

[241] Yu, X.; Chen, B.; Wang, Z.; et al.: Mobihydra: Pragmatic and multi-level plausibly deniable encryption storage for mobile devices. In *International Conference on Information Security*. Springer. 2014. pp. 555–567.

[242] Yue, Y.; Guo, L.; et al.: UNIX File System. In *UNIX Operating System*. Springer. 2011. pp. 149–185.

[243] Zandbergen, P. A.; Barbeau, S. J.: Positional accuracy of assisted gps data from high-sensitivity gps-enabled mobile phones. *Journal of Navigation*. vol. 64, no. 03. 2011: pp. 381–399.

[244] Zettel, D.-I. J.: Dynamic Linking in C++ under SunOS 4.1, Solaris 2.4 and Linux 1.3. 1996.

[245] Zhao, Z.; Osono, F. C. C.: "TrustDroid„: Preventing the use of SmartPhones for information leaking in corporate networks through the used of static analysis taint tracking. In *Malicious and Unwanted Software (MALWARE), 2012 7th International Conference on*. IEEE. 2012. pp. 135–143.

[246] Zhi-An, Y.; Chun-Miao, M.: The development and application of sensor based on android. In *Information Science and Digital Content Technology (ICIDT), 2012 8th International Conference on*, vol. 1. IEEE. 2012. pp. 231–234.

[247] Zhu, H.; Xiong, H.; Ge, Y.; et al.: Mobile app recommendations with security and privacy awareness. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM. 2014. pp. 951–960.

Appendices

Appendix A

Pseudocodes

```
1  /*
2   * On 64-bit platforms where user code may run
3   * in 32-bits the driver must
4   * translate the buffer (and local binder) addresses
5   * apropriately.
6   */
7
8  struct binder_write_read {
9          /* bytes to write */
10         signed long     write_size;
11         /* bytes consumed by driver */
12         signed long     write_consumed;
13         unsigned long   write_buffer;
14         /* bytes to read */
15         signed long     read_size;
16         /* bytes consumed by driver */
17         signed long     read_consumed;
18         unsigned long   read_buffer;
19 };
```

Listing A.1: Pseudocode of `binder_write_read` structure (defined in `binder.h`)

```
1  struct binder_transaction_data {
2    /* The first two are only used for bcTRANSACTION
3     * and brTRANSACTION,
4     * identifying the target and contents
5     * of the transaction.
6     */
7    union {
8      /* target descriptor of command transaction */
9      size_t handle;
10     /* target descriptor of return transaction */
11     void *ptr;
12   } target;
13
14   void *cookie; /* target object cookie */
15   unsigned int code;            /* transaction command */
16
17   /* General information about the transaction. */
18   unsigned int flags;
```

```
19    pid_t sender_pid;
20    uid_t sender_euid;
21    /* number of bytes of data */
22    size_t data_size;
23    /* number of bytes of offsets */
24    size_t offsets_size;
25
26    /* If this transaction is inline, the data immediately
27     * follows here; otherwise, it ends with a pointer to
28     * the data buffer.
29     */
30    union {
31      struct {
32        /* transaction data */
33        const void *buffer;
34        /* offsets from buffer to flat_binder_object
35         * structs
36         */
37        const void        *offsets;
38      } ptr;
39        uint8_t buf[8];
40    } data;
41  };
```

Listing A.2: Pseudocode of `binder_transaction_data` structure (defined in `binder.h`)

```
1  class SmallBlock {
2    unsigned long start; // Start Address
3    char block[SMALL_BLOCK_SIZE];
4  };
5
6  class MemBlock {
7    unsigned long start; // Start Address
8    unsigned char hash[32]; // Counted Hash Value
9    int size; // Size of Memory Block
10   int fdSrc; // Source File Descriptor
11   TaintedFile fpSrc; // Full Path of Source File
12 };
13
14 // List of Tainted Memory Block
15 std::vector<MemBlock *> taintedList;
16 // Blocks of Stored Information
17 std::vector<SmallBlock *> smallBlocks;
```

Listing A.3: Pseudocode of data structures for tainting mechanism

Appendix B

Source Code of Verification Models

```
<declaration>
  // constants of command type
  const int open_public   = 1;
  const int open_private  = 2;
  const int read_public   = 3;
  const int read_private  = 4;
  const int write_public  = 5;
  const int write_private = 6;
  const int share_content = 7;
  const int seek_position = 8;
  const int copy          = 9;
  const int close_public  = 10;
  const int close_private = 11;

  // shared command variable
  int command = 0;

  // synchronization channels
  chan read_command;
  chan next_command;
  chan user_action;
  broadcast chan read;
</declaration>
```

Listing B.1: Uppaal source code of variables definition

```
<template>
    <name x="5" y="5">FSMRequired</name>
    <declaration></declaration>
    <location id="id23" x="-528" y="-160">
        <name x="-538" y="-190">Cpr</name>
    </location>
    <location id="id24" x="-808" y="-160">
        <name x="-818" y="-190">Spr</name>
    </location>
    <location id="id25" x="-520" y="-280">
        <name x="-536" y="-264">Cpu</name>
    </location>
    <location id="id26" x="-808" y="-280">
        <name x="-824" y="-264">Spu</name>
```

```
15      </location>
16      <location id="id27" x="-976" y="-232">
17          <name x="-986" y="-262">s</name>
18      </location>
19      <init ref="id27" />
20      <transition>
21          <source ref="id26" />
22          <target ref="id26" />
23          <label kind="guard" x="-896" y="-416">
24              command == read_public ||
25              command == write_public ||
26              command == share_content
27          </label>
28          <label kind="synchronisation" x="-824" y="-368">
29              read?
30          </label>
31          <nail x="-776" y="-344" />
32          <nail x="-848" y="-344" />
33      </transition>
34      <transition>
35          <source ref="id24" />
36          <target ref="id24" />
37          <label kind="guard" x="-920" y="-88">
38              command == read_private ||
39              command == write_private
40          </label>
41          <label kind="synchronisation" x="-840" y="-56">
42              read?
43          </label>
44          <nail x="-840" y="-88" />
45          <nail x="-776" y="-88" />
46      </transition>
47      <transition>
48          <source ref="id25" />
49          <target ref="id26" />
50          <label kind="guard" x="-712" y="-360">
51              command == open_public
52          </label>
53          <label kind="synchronisation" x="-648" y="-336">
54              read?
55          </label>
56          <nail x="-528" y="-336" />
57          <nail x="-736" y="-336" />
58      </transition>
59      <transition>
60          <source ref="id23" />
61          <target ref="id24" />
62          <label kind="guard" x="-712" y="-120">
63              command == open_private
64          </label>
65          <label kind="synchronisation" x="-656" y="-104">
66              read?
67          </label>
68          <nail x="-544" y="-120" />
```

```
69          <nail x="-736" y="-120" />
70      </transition>
71      <transition>
72          <source ref="id26" />
73          <target ref="id25" />
74          <label kind="guard" x="-752" y="-304">
75              command == close_public
76          </label>
77          <label kind="synchronisation" x="-688" y="-280">
78              read?
79          </label>
80          <nail x="-672" y="-280" />
81      </transition>
82      <transition>
83          <source ref="id24" />
84          <target ref="id23" />
85          <label kind="guard" x="-760" y="-184">
86              command == close_private
87          </label>
88          <label kind="synchronisation" x="-688" y="-160">
89              read?
90          </label>
91          <nail x="-680" y="-160" />
92      </transition>
93      <transition>
94          <source ref="id27" />
95          <target ref="id24" />
96          <label kind="guard" x="-1008" y="-160">
97              command == open_private
98          </label>
99          <label kind="synchronisation" x="-928" y="-184">
100             read?
101         </label>
102         <nail x="-976" y="-160" />
103     </transition>
104     <transition>
105         <source ref="id27" />
106         <target ref="id26" />
107         <label kind="guard" x="-1024" y="-304">
108             command == open_public
109         </label>
110         <label kind="synchronisation" x="-944" y="-280">
111             read?
112         </label>
113         <nail x="-976" y="-280" />
114     </transition>
115 </template>
```

Listing B.2: Uppaal source code of FSMRequiredProcess

```
1  <template>
2      <name x="5" y="5">FSMImplementation</name>
3      <declaration></declaration>
4      <location id="id0" x="-112" y="112">
5          <name x="-120" y="128">Cpr</name>
```

```
6       </location>
7       <location id="id1" x="192" y="-8">
8           <name x="176" y="-40">Wpr</name>
9       </location>
10      <location id="id2" x="-112" y="-8">
11          <name x="-120" y="-40">Spr</name>
12      </location>
13      <location id="id3" x="-112" y="-232">
14          <name x="-128" y="-264">Cpu</name>
15      </location>
16      <location id="id4" x="200" y="-112">
17          <name x="184" y="-96">Wpu</name>
18      </location>
19      <location id="id5" x="-112" y="-112">
20          <name x="-120" y="-96">Spu</name>
21      </location>
22      <location id="id6" x="-288" y="-64">
23          <name x="-312" y="-72">s</name>
24      </location>
25      <init ref="id6" />
26      <transition>
27          <source ref="id1" />
28          <target ref="id1" />
29          <label kind="guard" x="272" y="-40">
30              command == read_private ||
31              command == write_private ||
32              command == seek_position
33          </label>
34          <label kind="synchronisation" x="328" y="8">
35              read?
36          </label>
37          <nail x="232" y="32" />
38          <nail x="272" y="-8" />
39          <nail x="232" y="-48" />
40      </transition>
41      <transition>
42          <source ref="id2" />
43          <target ref="id0" />
44          <label kind="guard" x="-104" y="40">
45              command == close_private
46          </label>
47          <label kind="synchronisation" x="-48" y="56">
48              read?
49          </label>
50      </transition>
51      <transition>
52          <source ref="id5" />
53          <target ref="id3" />
54          <label kind="guard" x="-104" y="-192">
55              command == close_public
56          </label>
57          <label kind="synchronisation" x="-56" y="-176">
58              read?
59          </label>
```

```
60      </transition>
61      <transition>
62          <source ref="id4" />
63          <target ref="id4" />
64          <label kind="guard" x="200" y="-248">
65              command == share_content ||
66              command == read_public ||
67              command == write_public ||
68              command == write_private ||
69              command == copy ||
70              command == seek_position
71          </label>
72          <label kind="synchronisation" x="264" y="-152">
73              read?
74          </label>
75          <nail x="248" y="-80" />
76          <nail x="272" y="-120" />
77          <nail x="240" y="-152" />
78      </transition>
79      <transition>
80          <source ref="id3" />
81          <target ref="id5" />
82          <label kind="guard" x="-336" y="-256">
83              command == open_public
84          </label>
85          <label kind="synchronisation" x="-272" y="-240">
86              read?
87          </label>
88          <nail x="-200" y="-232" />
89          <nail x="-200" y="-176" />
90      </transition>
91      <transition>
92          <source ref="id0" />
93          <target ref="id2" />
94          <label kind="guard" x="-352" y="88">
95              command == open_private
96          </label>
97          <label kind="synchronisation" x="-280" y="104">
98              read?
99          </label>
100         <nail x="-184" y="112" />
101         <nail x="-184" y="48" />
102     </transition>
103     <transition>
104         <source ref="id1" />
105         <target ref="id0" />
106         <label kind="guard" x="-20" y="110">
107             command == close_private
108         </label>
109         <label kind="synchronisation" x="40" y="128">
110             read?
111         </label>
112         <nail x="168" y="112" />
113     </transition>
```

```
114     <transition>
115         <source ref="id2" />
116         <target ref="id1" />
117         <label kind="guard" x="-56" y="-40">
118             command == read_private ||
119             command == write_private
120         </label>
121         <label kind="synchronisation" x="16" y="-8">
122             read?
123         </label>
124     </transition>
125     <transition>
126         <source ref="id4" />
127         <target ref="id3" />
128         <label kind="guard" x="-40" y="-256">
129             command == close_public
130         </label>
131         <label kind="synchronisation" x="16" y="-232">
132             read?
133         </label>
134         <nail x="120" y="-232" />
135     </transition>
136     <transition>
137         <source ref="id5" />
138         <target ref="id4" />
139         <label kind="guard" x="-48" y="-144">
140             command == read_public ||
141             command == write_public
142         </label>
143         <label kind="synchronisation" x="8" y="-112">
144             read?
145         </label>
146     </transition>
147     <transition>
148         <source ref="id6" />
149         <target ref="id2" />
150         <label kind="guard" x="-312" y="-8">
151             command == open_private
152         </label>
153         <label kind="synchronisation" x="-248" y="8">
154             read?
155         </label>
156         <nail x="-288" y="-8" />
157     </transition>
158     <transition>
159         <source ref="id6" />
160         <target ref="id5" />
161         <label kind="guard" x="-312" y="-136">
162             command == open_public
163         </label>
164         <label kind="synchronisation" x="-248" y="-112">
165             read?
166         </label>
167         <nail x="-288" y="-112" />
```

```
168     </transition>
169 </template>
```

Listing B.3: Uppaal source code of FSMImplementationProcess

```
 1 <template>
 2     <name x="5" y="5">User</name>
 3     <declaration></declaration>
 4     <location id="id11" x="-824" y="-368">
 5         <name x="-834" y="-398">i</name>
 6     </location>
 7     <location id="id12" x="-720" y="-208">
 8         <name x="-730" y="-238">h</name>
 9     </location>
10     <location id="id13" x="-704" y="-128">
11         <name x="-714" y="-158">g</name>
12     </location>
13     <location id="id14" x="-992" y="-128">
14         <name x="-1002" y="-158">f</name>
15     </location>
16     <location id="id15" x="-1256" y="-128">
17         <name x="-1266" y="-158">e</name>
18     </location>
19     <location id="id16" x="-1200" y="-200">
20         <name x="-1210" y="-230">d</name>
21     </location>
22     <location id="id17" x="-760" y="-424">
23         <name x="-770" y="-454">j</name>
24     </location>
25     <location id="id18" x="-704" y="-480">
26         <name x="-714" y="-510">k</name>
27     </location>
28     <location id="id19" x="-1256" y="-480">
29         <name x="-1266" y="-510">c</name>
30     </location>
31     <location id="id20" x="-1216" y="-424">
32         <name x="-1226" y="-454">b</name>
33     </location>
34     <location id="id21" x="-1152" y="-368">
35         <name x="-1162" y="-398">a</name>
36     </location>
37     <location id="id22" x="-992" y="-328">
38         <name x="-1002" y="-358">s</name>
39     </location>
40     <init ref="id22" />
41     <transition>
42         <source ref="id21" />
43         <target ref="id22" />
44         <label kind="synchronisation" x="-1136" y="-336">
45             read?
46         </label>
47         <nail x="-1152" y="-312" />
48     </transition>
49     <transition>
50         <source ref="id20" />
```

```
        <target ref="id22" />
        <label kind="synchronisation" x="-1208" y="-320">
            read?
        </label>
        <nail x="-1216" y="-288" />
    </transition>
    <transition>
        <source ref="id19" />
        <target ref="id22" />
        <label kind="synchronisation" x="-1248" y="-288">
            read?
        </label>
        <nail x="-1256" y="-256" />
    </transition>
    <transition>
        <source ref="id18" />
        <target ref="id22" />
        <label kind="synchronisation" x="-752" y="-304">
            read?
        </label>
        <nail x="-704" y="-264" />
    </transition>
    <transition>
        <source ref="id17" />
        <target ref="id22" />
        <label kind="synchronisation" x="-800" y="-336">
            read?
        </label>
        <nail x="-760" y="-304" />
    </transition>
    <transition>
        <source ref="id11" />
        <target ref="id22" />
        <label kind="synchronisation" x="-896" y="-352">
            read?
        </label>
        <nail x="-824" y="-328" />
    </transition>
    <transition>
        <source ref="id12" />
        <target ref="id22" />
        <label kind="synchronisation" x="-880" y="-264">
            read?
        </label>
    </transition>
    <transition>
        <source ref="id13" />
        <target ref="id22" />
        <label kind="synchronisation" x="-752" y="-256">
            read?
        </label>
        <nail x="-704" y="-256" />
    </transition>
    <transition>
```

```
        <source ref="id14" />
        <target ref="id22" />
        <label kind="synchronisation" x="-984" y="-208">
            read?
        </label>
    </transition>
    <transition>
        <source ref="id15" />
        <target ref="id22" />
        <label kind="synchronisation" x="-1248" y="-184">
            read?
        </label>
        <nail x="-1256" y="-232" />
    </transition>
    <transition>
        <source ref="id16" />
        <target ref="id22" />
        <label kind="synchronisation" x="-1072" y="-288">
            read?
        </label>
        <nail x="-1176" y="-240" />
    </transition>
    <transition>
        <source ref="id22" />
        <target ref="id11" />
        <label kind="synchronisation" x="-920" y="-392">
            user_action!
        </label>
        <label kind="assignment" x="-944" y="-408">
            command = write_private
        </label>
        <nail x="-944" y="-368" />
    </transition>
    <transition>
        <source ref="id22" />
        <target ref="id12" />
        <label kind="synchronisation" x="-848" y="-192">
            user_action!
        </label>
        <label kind="assignment" x="-896" y="-208">
            command = seek_position
        </label>
        <nail x="-896" y="-208" />
    </transition>
    <transition>
        <source ref="id22" />
        <target ref="id13" />
        <label kind="synchronisation" x="-784" y="-104">
            user_action!
        </label>
        <label kind="assignment" x="-800" y="-120">
            command = copy
        </label>
        <nail x="-864" y="-128" />
```

```
    </transition>
    <transition>
        <source ref="id22" />
        <target ref="id14" />
        <label kind="synchronisation" x="-984" y="-104">
            user_action!
        </label>
        <label kind="assignment" x="-1024" y="-120">
            command = share_content
        </label>
        <nail x="-888" y="-128" />
    </transition>
    <transition>
        <source ref="id22" />
        <target ref="id15" />
        <label kind="synchronisation" x="-1168" y="-152">
            user_action!
        </label>
        <label kind="assignment" x="-1200" y="-168">
            command = close_public
        </label>
        <nail x="-1016" y="-128" />
    </transition>
    <transition>
        <source ref="id22" />
        <target ref="id16" />
        <label kind="synchronisation" x="-1144" y="-224">
            user_action!
        </label>
        <label kind="assignment" x="-1176" y="-240">
            command = close_private
        </label>
        <nail x="-1016" y="-200" />
    </transition>
    <transition>
        <source ref="id22" />
        <target ref="id17" />
        <label kind="synchronisation" x="-872" y="-448">
            user_action!
        </label>
        <label kind="assignment" x="-904" y="-464">
            command = write_public
        </label>
        <nail x="-936" y="-424" />
    </transition>
    <transition>
        <source ref="id22" />
        <target ref="id18" />
        <label kind="synchronisation" x="-864" y="-504">
            user_action!
        </label>
        <label kind="assignment" x="-904" y="-520">
            command = read_private
        </label>
```

```
213         <nail x="-944" y="-480" />
214     </transition>
215     <transition>
216         <source ref="id22" />
217         <target ref="id19" />
218         <label kind="synchronisation" x="-1152" y="-504">
219             user_action!
220         </label>
221         <label kind="assignment" x="-1184" y="-520">
222             command = read_public
223         </label>
224         <nail x="-976" y="-480" />
225     </transition>
226     <transition>
227         <source ref="id22" />
228         <target ref="id20" />
229         <label kind="synchronisation" x="-1152" y="-448">
230             user_action!
231         </label>
232         <label kind="assignment" x="-1176" y="-464">
233             command = open_private
234         </label>
235         <nail x="-992" y="-424" />
236     </transition>
237     <transition>
238         <source ref="id22" />
239         <target ref="id21" />
240         <label kind="synchronisation" x="-1120" y="-392">
241             user_action!
242         </label>
243         <label kind="assignment" x="-1160" y="-408">
244             command = open_public
245         </label>
246         <nail x="-1096" y="-368" />
247     </transition>
248 </template>
```

Listing B.4: Uppaal source code of user process

```
1  <template>
2      <name x="5" y="5">UserPublic</name>
3      <declaration></declaration>
4      <location id="id28" x="-1000" y="-528">
5          <name x="-1010" y="-558">a</name>
6      </location>
7      <location id="id29" x="-1224" y="-528">
8          <name x="-1234" y="-558">s</name>
9      </location>
10     <location id="id30" x="-864" y="-440">
11         <name x="-874" y="-470">i</name>
12     </location>
13     <location id="id31" x="-864" y="-352">
14         <name x="-874" y="-382">h</name>
15     </location>
16     <location id="id32" x="-968" y="-312">
```

```
17          <name x="-978" y="-342">g</name>
18      </location>
19      <location id="id33" x="-1192" y="-312">
20          <name x="-1202" y="-342">f</name>
21      </location>
22      <location id="id34" x="-1280" y="-376">
23          <name x="-1304" y="-384">e</name>
24      </location>
25      <location id="id35" x="-992" y="-480">
26          <name x="-1002" y="-510">j</name>
27      </location>
28      <location id="id36" x="-1184" y="-472">
29          <name x="-1194" y="-502">c</name>
30      </location>
31      <location id="id37" x="-1280" y="-408">
32          <name x="-1304" y="-424">d</name>
33      </location>
34      <location id="id38" x="-1080" y="-408">
35          <name x="-1090" y="-438">b</name>
36      </location>
37      <init ref="id29" />
38      <transition>
39          <source ref="id28" />
40          <target ref="id38" />
41          <label kind="synchronisation" x="-1008" y="-520">
42              read?
43          </label>
44      </transition>
45      <transition>
46          <source ref="id29" />
47          <target ref="id28" />
48          <label kind="synchronisation" x="-1152" y="-552">
49              user_action!
50          </label>
51          <label kind="assignment" x="-1192" y="-568">
52              command = open_public
53          </label>
54      </transition>
55      <transition>
56          <source ref="id30" />
57          <target ref="id38" />
58          <label kind="synchronisation" x="-952" y="-432">
59              read?
60          </label>
61      </transition>
62      <transition>
63          <source ref="id38" />
64          <target ref="id30" />
65          <label kind="synchronisation" x="-840" y="-408">
66              user_action!
67          </label>
68          <label kind="assignment" x="-856" y="-424">
69              command = write_private
70          </label>
```

```
71          <nail x="-864" y="-384" />
72          <nail x="-864" y="-416" />
73      </transition>
74      <transition>
75          <source ref="id37" />
76          <target ref="id38" />
77          <label kind="synchronisation" x="-1240" y="-432">
78              read?
79          </label>
80      </transition>
81      <transition>
82          <source ref="id36" />
83          <target ref="id38" />
84          <label kind="synchronisation" x="-1128" y="-456">
85              read?
86          </label>
87      </transition>
88      <transition>
89          <source ref="id35" />
90          <target ref="id38" />
91          <label kind="synchronisation" x="-1016" y="-464">
92              read?
93          </label>
94      </transition>
95      <transition>
96          <source ref="id31" />
97          <target ref="id38" />
98          <label kind="synchronisation" x="-936" y="-360">
99              read?
100         </label>
101     </transition>
102     <transition>
103         <source ref="id32" />
104         <target ref="id38" />
105         <label kind="synchronisation" x="-1048" y="-352">
106             read?
107         </label>
108     </transition>
109     <transition>
110         <source ref="id33" />
111         <target ref="id38" />
112         <label kind="synchronisation" x="-1144" y="-360">
113             read?
114         </label>
115     </transition>
116     <transition>
117         <source ref="id34" />
118         <target ref="id38" />
119         <label kind="synchronisation" x="-1240" y="-384">
120             read?
121         </label>
122     </transition>
123     <transition>
124         <source ref="id38" />
```

```
125         <target ref="id31" />
126         <label kind="synchronisation" x="-840" y="-328">
127             user_action!
128         </label>
129         <label kind="assignment" x="-856" y="-344">
130             command = seek_position
131         </label>
132         <nail x="-864" y="-296" />
133     </transition>
134     <transition>
135         <source ref="id38" />
136         <target ref="id32" />
137         <label kind="synchronisation" x="-1040" y="-288">
138             user_action!
139         </label>
140         <label kind="assignment" x="-1056" y="-304">
141             command = copy
142         </label>
143         <nail x="-1072" y="-312" />
144     </transition>
145     <transition>
146         <source ref="id38" />
147         <target ref="id33" />
148         <label kind="synchronisation" x="-1168" y="-288">
149             user_action!
150         </label>
151         <label kind="assignment" x="-1232" y="-304">
152             command = share_content
153         </label>
154         <nail x="-1096" y="-312" />
155     </transition>
156     <transition>
157         <source ref="id38" />
158         <target ref="id34" />
159         <label kind="synchronisation" x="-1352" y="-312">
160             user_action!
161         </label>
162         <label kind="assignment" x="-1392" y="-328">
163             command = close_public
164         </label>
165         <nail x="-1280" y="-328" />
166     </transition>
167     <transition>
168         <source ref="id38" />
169         <target ref="id35" />
170         <label kind="synchronisation" x="-904" y="-488">
171             user_action!
172         </label>
173         <label kind="assignment" x="-936" y="-504">
174             command = write_public
175         </label>
176         <nail x="-920" y="-448" />
177         <nail x="-920" y="-480" />
178     </transition>
```

```
179     <transition>
180         <source ref="id38" />
181         <target ref="id36" />
182         <label kind="synchronisation" x="-1152" y="-496">
183             user_action!
184         </label>
185         <label kind="assignment" x="-1184" y="-512">
186             command = read_public
187         </label>
188         <nail x="-1080" y="-472" />
189     </transition>
190     <transition>
191         <source ref="id38" />
192         <target ref="id37" />
193         <label kind="synchronisation" x="-1320" y="-488">
194             user_action!
195         </label>
196         <label kind="assignment" x="-1352" y="-504">
197             command = open_public
198         </label>
199         <nail x="-1280" y="-472" />
200     </transition>
201 </template>
```

Listing B.5: Uppaal source code of user process limited to public file operations

```
 1 <template>
 2     <name x="5" y="5">UserPrivate</name>
 3     <declaration>// Place local declarations here.
 4 </declaration>
 5     <location id="id39" x="-1920" y="-280">
 6         <name x="-1930" y="-310">a</name>
 7     </location>
 8     <location id="id40" x="-2112" y="-280">
 9         <name x="-2122" y="-310">s</name>
10     </location>
11     <location id="id41" x="-1672" y="-352">
12         <name x="-1682" y="-382">d</name>
13     </location>
14     <location id="id42" x="-1864" y="-312">
15         <name x="-1874" y="-342">e</name>
16     </location>
17     <location id="id43" x="-2016" y="-320">
18         <name x="-2026" y="-350">f</name>
19     </location>
20     <location id="id44" x="-1728" y="-464">
21         <name x="-1738" y="-494">c</name>
22     </location>
23     <location id="id45" x="-2016" y="-464">
24         <name x="-2026" y="-494">g</name>
25     </location>
26     <location id="id46" x="-1864" y="-408">
27         <name x="-1874" y="-438">b</name>
28     </location>
29     <init ref="id40" />
```

```
30      <transition>
31          <source ref="id39" />
32          <target ref="id46" />
33          <label kind="synchronisation" x="-1944" y="-336">
34              read?
35          </label>
36      </transition>
37      <transition>
38          <source ref="id40" />
39          <target ref="id39" />
40          <label kind="synchronisation" x="-2064" y="-264">
41              user_action!
42          </label>
43          <label kind="assignment" x="-2096" y="-280">
44              command = open_private
45          </label>
46      </transition>
47      <transition>
48          <source ref="id45" />
49          <target ref="id46" />
50          <label kind="synchronisation" x="-1984" y="-440">
51              read?
52          </label>
53      </transition>
54      <transition>
55          <source ref="id44" />
56          <target ref="id46" />
57          <label kind="synchronisation" x="-1816" y="-464">
58              read?
59          </label>
60      </transition>
61      <transition>
62          <source ref="id41" />
63          <target ref="id46" />
64          <label kind="synchronisation" x="-1736" y="-360">
65              read?
66          </label>
67      </transition>
68      <transition>
69          <source ref="id42" />
70          <target ref="id46" />
71          <label kind="synchronisation" x="-1856" y="-368">
72              read?
73          </label>
74      </transition>
75      <transition>
76          <source ref="id43" />
77          <target ref="id46" />
78          <label kind="synchronisation" x="-1936" y="-368">
79              read?
80          </label>
81      </transition>
82      <transition>
83          <source ref="id46" />
```

```
84          <target ref="id41" />
85          <label kind="synchronisation" x="-1664" y="-416">
86              user_action!
87          </label>
88          <label kind="assignment" x="-1744" y="-432">
89              command = write_private
90          </label>
91          <nail x="-1672" y="-408" />
92      </transition>
93      <transition>
94          <source ref="id46" />
95          <target ref="id42" />
96          <label kind="synchronisation" x="-1792" y="-296">
97              user_action!
98          </label>
99          <label kind="assignment" x="-1832" y="-312">
100             command = seek_position
101         </label>
102         <nail x="-1776" y="-368" />
103         <nail x="-1776" y="-312" />
104     </transition>
105     <transition>
106         <source ref="id46" />
107         <target ref="id43" />
108         <label kind="synchronisation" x="-2136" y="-368">
109             user_action!
110         </label>
111         <label kind="assignment" x="-2184" y="-384">
112             command = close_private
113         </label>
114         <nail x="-2016" y="-400" />
115     </transition>
116     <transition>
117         <source ref="id46" />
118         <target ref="id44" />
119         <label kind="synchronisation" x="-1832" y="-544">
120             user_action!
121         </label>
122         <label kind="assignment" x="-1864" y="-560">
123             command = read_private
124         </label>
125         <nail x="-1856" y="-520" />
126         <nail x="-1728" y="-520" />
127     </transition>
128     <transition>
129         <source ref="id46" />
130         <target ref="id45" />
131         <label kind="synchronisation" x="-2016" y="-544">
132             user_action!
133         </label>
134         <label kind="assignment" x="-2056" y="-560">
135             command = open_private
136         </label>
137         <nail x="-1944" y="-520" />
```

```
138         <nail x="-2016" y="-520" />
139     </transition>
140 </template>
```

Listing B.6: Uppaal source code of user process limited to private file operations

```
 1 <template>
 2     <name x="5" y="5">Reader</name>
 3     <declaration></declaration>
 4     <location id="id7" x="-840" y="-232">
 5         <name x="-840" y="-216">a</name>
 6     </location>
 7     <location id="id8" x="-840" y="-344">
 8         <name x="-850" y="-374">s</name>
 9     </location>
10     <init ref="id8" />
11     <transition>
12         <source ref="id7" />
13         <target ref="id8" />
14         <label kind="synchronisation" x="-800" y="-304">
15                 read!
16         </label>
17         <nail x="-808" y="-288" />
18     </transition>
19     <transition>
20         <source ref="id8" />
21         <target ref="id7" />
22         <label kind="synchronisation" x="-976" y="-304">
23                 next_command?
24         </label>
25         <nail x="-872" y="-288" />
26     </transition>
27 </template>
```

Listing B.7: Uppaal source code of reader

```
 1 <template>
 2     <name x="5" y="5">Writer</name>
 3     <declaration></declaration>
 4     <location id="id9" x="-1000" y="-416">
 5         <name x="-1000" y="-400">a</name>
 6     </location>
 7     <location id="id10" x="-1000" y="-520">
 8         <name x="-1010" y="-550">s</name>
 9     </location>
10     <init ref="id10" />
11     <transition>
12         <source ref="id9" />
13         <target ref="id10" />
14         <label kind="synchronisation" x="-968" y="-488">
15             next_command!
16         </label>
17         <nail x="-968" y="-472" />
18     </transition>
19     <transition>
```

```
20          <source ref="id10" />
21          <target ref="id9" />
22          <label kind="synchronisation" x="-1120" y="-488">
23              user_action?
24          </label>
25          <nail x="-1032" y="-472" />
26      </transition>
27  </template>
```

Listing B.8: Uppaal source code of writer

```
1  <system>
2    FSMImplementationProcess = FSMImplementation();
3    FSMRequiredProcess = FSMRequired();
4    ReaderProcess = Reader();
5    WriterProcess = Writer();
6
7    // What User process will be used
8    UserProcess = User();
9    //UserProcess = UserPublic();
10   //UserProcess = UserPrivate();
11
12   // List one or more processes to be composed into a system.
13   system FSMImplementationProcess, FSMRequiredProcess,
14          ReaderProcess, WriterProcess, UserProcess;
15 </system>
```

Listing B.9: Uppaal source code of system definition