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

THESIS
TEZE DISERTAČNÍ PRÁCE

AUTHOR
AUTOR PRÁCE

Ing. LUKÁŠ ARON

SUPERVISOR
ŠKOLITEL

Doc. Dr. Ing. PETR HANÁČEK

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. 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. 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. Thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Hanáček Petr.

Privacy Protection on Mobile Devices

Declaration

Prohlašuji, že jsem tyto teze vypracoval samostatně pod vedením
pana Doc. Dr. Ing. Petra Hanáčka

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

....
Lukáš Aron
March 16, 2018

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 [42]. 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 [20]. 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 [4] or mobile device management (MDM) approach [8]. 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. The 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, thirdparty 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.

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.

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 [39]. 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.

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.

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. [40] use probabilistic generative models for risk scoring schemes, and identify several models ranging from the simple Naive Bayes models [43] to advanced hierarchical mixture models. Moreover, Zhu et al. [59] developed a recommendation system that considers both the application's popularity and the privacy threat. This solution is based on the work of [40]. Different from using permission to assess risk an automated framework called RISKMON was presented by [25]. 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 [50].

Monitoring run-time process is helpful for users to understand the privacy data flow and detect the malicious behavior. For instance, the FireDroid [44] 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. [49].

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 [3]. 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 [12] is NDroid [41] and it performs dynamic taint analysis. This system is designed for checking information flows through Java native interface (JNI) [17]. 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) [57]. 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 [32] 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 [13], 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 [18], while the underlying kernel is implemented by C programming language [26]. The similar approach is with the operating system iOS [23]. The kernel of iOS is also implemented by C programming language and applications are developed in Swift programming language [51]. 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 [56] is one of many malware analysis detection platform. It is based on top of quick emulator (QEMU) [5] (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.

2.1 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.

Chapter 3

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 [52].

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 3.1.

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 [38, 11] (or the successor of Dalvik called Android runtime (ART) virtual machine [7]) 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.

3.1 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-

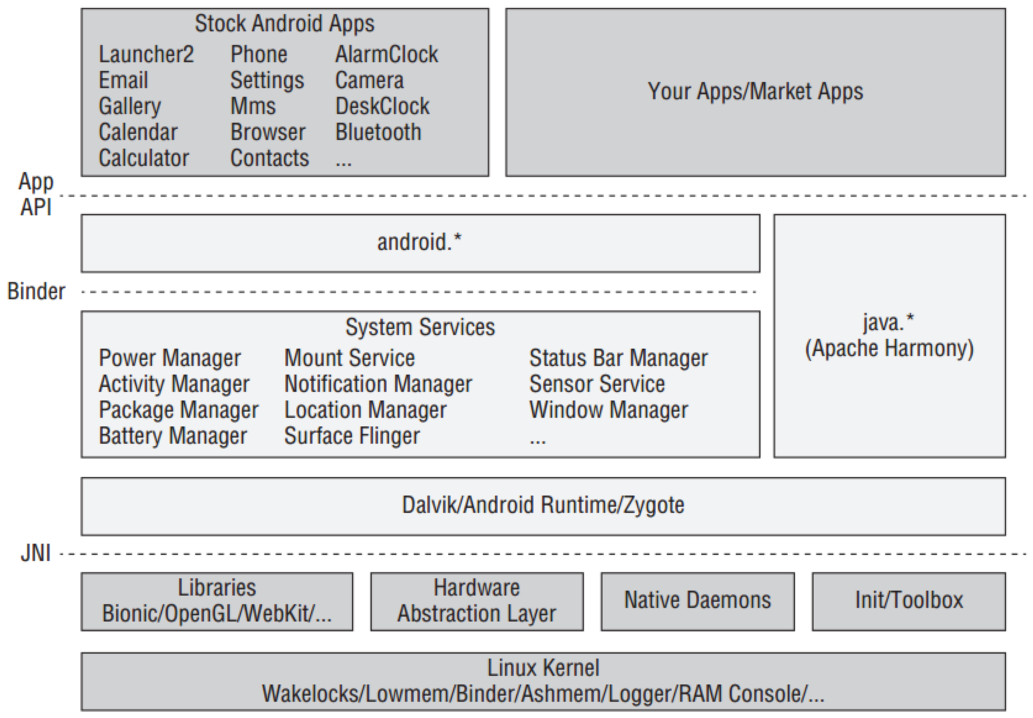

Figure 3.1: General Android system architecture [9]

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.

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.

3.2 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, i.e., each application has its memory space assigned. Additional security is achieved by utilizing memory management unit (MMU) [16], 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 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.

3.3 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 [19]. 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.

3.4 Architecture Levels

According to Android architecture presented in figure 3.1 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.

The Linux kernel enforces the isolation of applications and operating system components exploiting standard Linux facilities [37] (process separation and DAC [30] over network socket and file system). This isolation is imposed by assigning each application a separate user ID and group identifier (GID) [47], as was discussed in section 3.2. 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 [53]. 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) [14] 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 **??**) with a particular driver released at the kernel level.

Linux Kernel

In Android platform, Linux kernel [33] 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* [46] - 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 [55].

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.

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 [45]. Therefore, an inter-process communication (IPC) [1] framework is required to organize data and signals exchange between different processes. In Android, a special framework called *binder* [1, 46] is used for inter-process communication. The standard POSIX system V [29] 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 [48] 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 [46].

The communication between the processes is organized according to synchronous client-server model [6]. 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.

3.5 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.

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 4

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 [36]. 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).

4.1 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.

4.2 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 4.1 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.

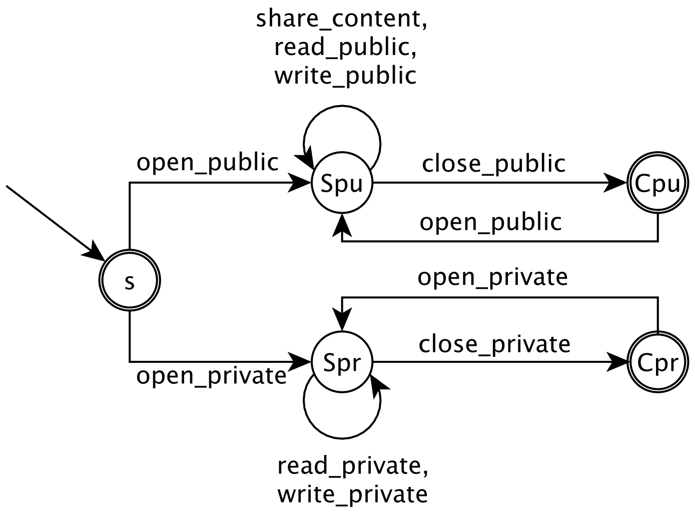

Figure 4.1: Required behavior on file level defined by finite state machine

Figure 4.1 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.
- 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 4.1 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 4.1.

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

Table 4.1: Required definition of state-transition function (δ) for FSM.

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 [21] - Turing machine (TM) was chosen. In order to define the formal model by the TM which simulates the FSM (see figure 4.1), the logic needs to be defined.

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 4.2.1.

Definition 4.2.1. Let $\mathbb{N}_+$ is a set of positive integer values, $F = \{F_1, F_2, \dots, F_n\}$ 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 4.1. After reading the symbol from the input tape (first tape of TM), the reading head moves to the right automatically.

$$\begin{aligned}\Sigma = I^+[&open\_public|open\_private|share\_content|read\_public|read\_private \\ &|write\_public|write\_private|close\_public|close\_private]\end{aligned} \tag{4.1}$$

This regular expression 4.1 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 4.1. 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 4.2.

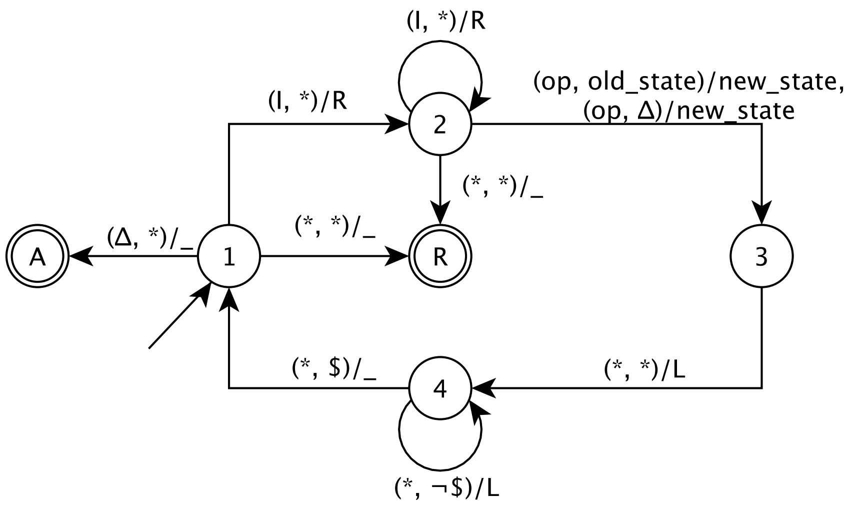

Figure 4.2: Turing machine models required behavior

Related to figure 4.2 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. 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 4.2.1. 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 4.2.

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

Table 4.2: 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.

4.3 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 5

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.

5.1 Framework

The implementation framework for the prototype is called Aurasium [54]. 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 [27]. 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.

5.2 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 5.1.

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

Table 5.1: 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 [15] is customized to provide inter-process communication as was described in the section 3.4. 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 3.4.

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 [22]. 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.

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. 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 automatically parses and performs read commands on the service side.

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.

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 **??**. 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 **??**), 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 5.1.

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 a start address and its size and is linked to one counted hash value, several small blocks

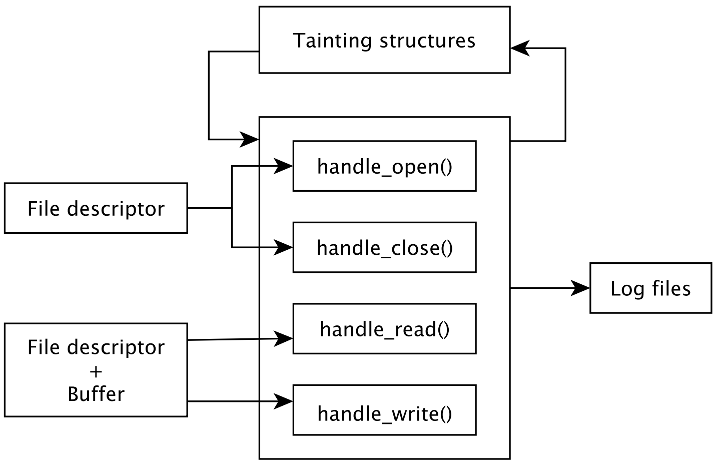

Figure 5.1: Overview of the proposed tainting system

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 [35] 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 5.2.1. 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 5.2.1. Let set $F = \{F_1, F_2, \dots, F_n\}$ is a set of private and tainted files, where n is the amount of theses files and $\forall x \in \{1, 2, \dots, bAmount(F_d)\} : b_x(F_d)$ is a Small Block of a file F_d, where $bAmount(F_d)$ is an amount of created Small Blocks for the file $F_d, d \in \{1, 2, \dots, n\}$ 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, $\forall d_1, d_2 \in \{1, 2, \dots, n\} \forall x \in \{1, 2, \dots, bAmount(F_{d1})\} \forall y \in \{1, 2, \dots, bAmount(F_{d2})\} : [(b_x(F_{d1})) = b_y(F_{d2})] \Rightarrow (F_{d1}, F_{d2}) \in R$.

In other words, the statement definition 5.2.1 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 5.2.

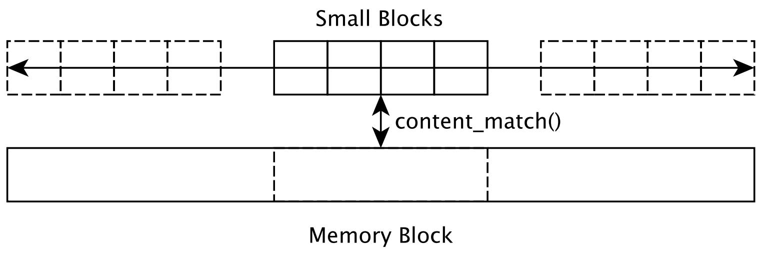

Figure 5.2: Principle of content-based scanning

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 **??**) and the transaction name is *QUERY_TRANSACTION*, then the hardened application's *query()* function is called. Illustration of this process is described in figure 5.3.

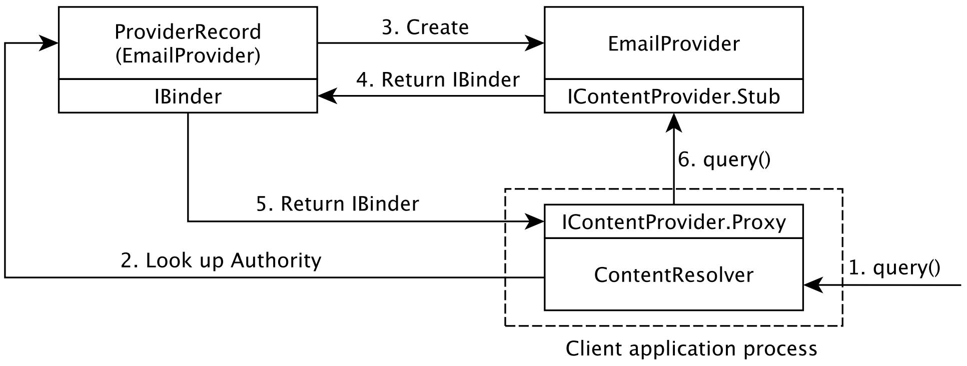

Figure 5.3: Content provider interaction

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.*

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.

5.3 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 4.2.1), 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 **??** 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 5.2.

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. 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.

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

Table 5.2: Definition of state-transition function (δ).

- $\Sigma \setminus \{\Delta\}$ - 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 4.2.1, 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 when the reading possible according to state-transition function.

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

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. The TM is defined in very similar format as the model of required behavior. Due to the lack of space the full description is described in the full version of the thesis.

5.4 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 6

Formal Verification

In the context of software systems, formal verification [34] 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.

6.1 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) [10], 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 [2], 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 [58, 28]. The verifier checks for simple invariants and reachability properties for efficiency reasons. Other properties may be checked by using testing automata [24] or the decorated system with debugging information [31].

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 6.1.

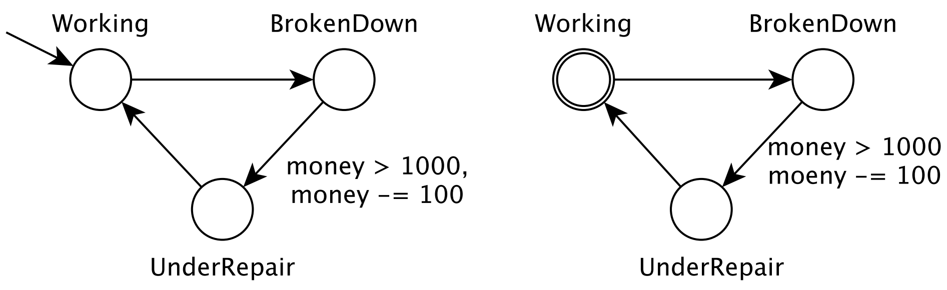

Figure 6.1: Example of Models with action

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 6.2. 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.

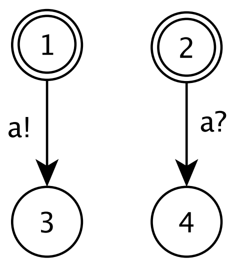

Figure 6.2: Simple synchronization example in Uppaal tool

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.

6.2 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.

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 6.3.

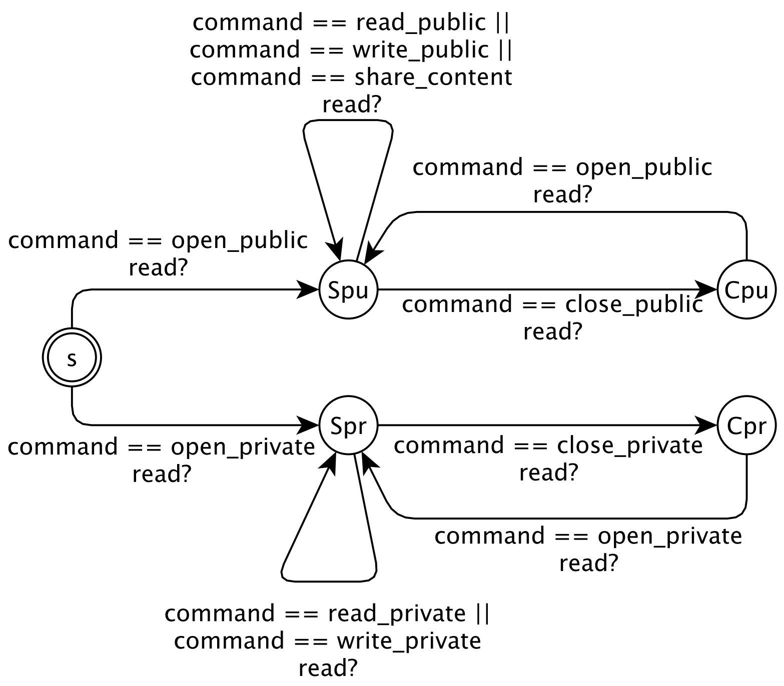

Figure 6.3: Model of required behavior in Uppaal format

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*.

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 6.4. 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.

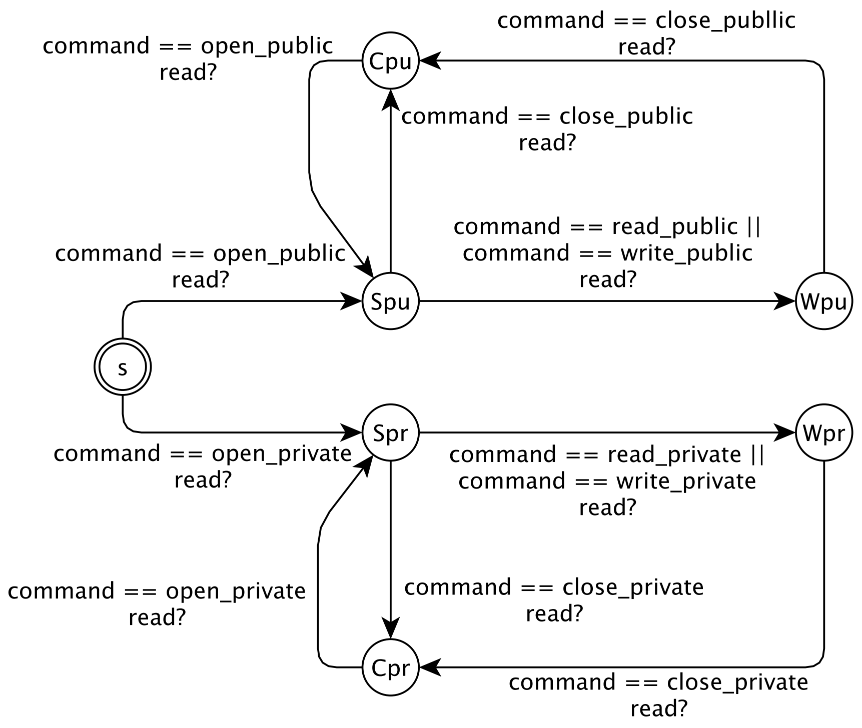

Figure 6.4: Model of implementation in Uppaal format

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.

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

Chapter 7

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).

There are two possible explanation before the experiment 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.

7.1 Experiment

The verification experiment 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*.

Moreover, model of required behavior is named *FSMRequiredProcess*, model of implementation is called *FSMImplementationProcess*. 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 7.1 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 7.1: Verification rule for initial states

Next rules depicted in listing 7.2 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 7.2: Verification rules for opened file

This experiment shows that the general required behavior of implementation method should be satisfied. The full version of the thesis contains three experiments and there are described the results of experiments, as well.

Chapter 8

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 4. 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 5. 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 6. 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. Example of verification process demonstrate the usability of the proposed method, presented in chapter 7.

Bibliography

[1] Aleksandar Gargenta: Deep Dive into Android IPC/Binder Framework. `https://thenewcircle.com/s/post/1340/Deep_Dive_Into_Binder_Presentation.htm`. 2012.

[2] Alur, R.; Dill, D. L.: A theory of timed automata. *Theoretical computer science*. vol. 126, no. 2. 1994: pp. 183–235.

[3] 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.

[4] 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.

[5] Bartholomew, D.: Qemu a multihost multitarget emulator. *Linux Journal*. vol. 2006, no. 145. 2006: page 3.

[6] Birman, K. P.: Remote Procedure Calls and the Client/Server Model. In *Guide to Reliable Distributed Systems*. Springer. 2012. pp. 185–247.

[7] Cinar, O.: Android Platform. In *Android Quick APIs Reference*. Springer. 2015. pp. 1–14.

[8] Danford, T. E.; Batchu, S. K.: Virtual instance architecture for mobile device management systems. November 15 2011. uS Patent 8,060,074.

[9] Drake, J. J.; Lanier, Z.; Mulliner, C.; et al.: *Android Hacker's Handbook*. John Wiley & Sons. 2014.

[10] Dumas, M.; Ter Hofstede, A. H.: UML activity diagrams as a workflow specification language. In *UML*, vol. 2185. Springer. 2001. pp. 76–90.

[11] Ehringer, D.: The dalvik virtual machine architecture. *Techn. report (March 2010)*. vol. 4. 2010: page 8.

[12] Enck, W.; Gilbert, P.; Han, S.; et al.: TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*. vol. 32, no. 2. 2014: page 5.

[13] Forman, I. R.; Forman, N.; Ibm, J. V.: Java reflection in action. 2004.

[14] Gargenta, A.: Deep dive into Android IPC/Binder framework. *Android Builders Summit*. 2013.

[15] Garrity, D. F.: Binding apparatus. May 16 2000. uS Patent 6,062,792.

[16] Gelter, A.; Parker, B.; Boatright, R.; et al.: Memory management unit. 2013. uS Patent 8,443,098.

[17] Gordon, R.: *Essential JNI: Java Native Interface*. Prentice-Hall, Inc.. 1998.

[18] Gosling, J.: *The Java language specification*. Addison-Wesley Professional. 2000.

[19] Götzfried, J.; Müller, T.: Analysing Android's Full Disk Encryption Feature. *JoWUA*. vol. 5, no. 1. 2014: pp. 84–100.

[20] Halpert, B.: Mobile device security. In *Proceedings of the 1st annual conference on Information security curriculum development*. ACM. 2004. pp. 99–101.

[21] Herken, R.: The Universal Turing Machine. A Half-Century Survey. 1992.

[22] 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.

[23] Hoog, A.; Strzempka, K.: *iPhone and iOS forensics: Investigation, analysis and mobile security for Apple iPhone, iPad and iOS devices*. Elsevier. 2011.

[24] 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.

[25] 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.

[26] Kernighan, B. W.; Ritchie, D. M.: *The C programming language*. 2006.

[27] 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.

[28] 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.

[29] Lewine, D.: *POSIX programmers guide*. „ O'Reilly Media, Inc.“. 1991.

[30] Li, N.: Discretionary access control. In *Encyclopedia of Cryptography and Security*. Springer. 2011. pp. 353–356.

[31] 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.

[32] 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.

[33] Mauerer, W.: *Professional Linux kernel architecture*. John Wiley & Sons. 2010.

[34] 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.

[35] Menezes, A. J.; Van Oorschot, P. C.; Vanstone, S. A.: *Handbook of applied cryptography*. CRC press. 1996.

[36] Miller, K. W.; Voas, J.; Hurlburt, G. F.: BYOD: security and privacy considerations. *It Professional*. vol. 14, no. 5. 2012: pp. 0053–55.

[37] Morris, J.; Smalley, S.; Kroah-Hartman, G.: Linux security modules: General security support for the linux kernel. In *USENIX Security Symposium*. 2002.

[38] 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.

[39] Papadimitriou, P.; Garcia-Molina, H.: Data leakage detection. *IEEE Transactions on knowledge and data engineering*. vol. 23, no. 1. 2011: pp. 51–63.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] Sandhu, R. S.; Samarati, P.: Access control: principle and practice. *Communications Magazine, IEEE*. vol. 32, no. 9. 1994: pp. 40–48.

[46] 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.

[47] 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.

[48] Stevens, W. R.; Fenner, B.; Rudoff, A. M.: *UNIX network programming.* vol. 1. Addison-Wesley Professional. 2004.

[49] 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.

[50] 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.

[51] Wells, G.: The Future of iOS Development: Evaluating the Swift Programming Language. 2015.

[52] 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.

[53] WWW Pages: Android Security Overview. `https://source.android.com/security/`. 2017.

[54] Xu, R.; Saïdi, H.; Anderson, R. J.: Aurasium: practical policy enforcement for android applications. In *USENIX Security Symposium*, vol. 2012. 2012.

[55] Yaghmour, K.: *Embedded Android: Porting, Extending, and Customizing.* „ O'Reilly Media, Inc.“. 2013.

[56] 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.

[57] 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.

[58] 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.

[59] 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.