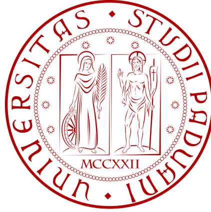

UNIVERSITY OF PADOVA

Department of Information Engineering

Master Degree in Computer Engineering

MASTER THESIS

UserLoop: User-in-the-Loop verification for Privacy Protection in Mobile Applications

Supervisor
Prof. Mauro Conti
University of Padova
Department of Mathematics

Co-supervisor
Prof. Giovanni Russello
University of Auckland
Department of Computer Science

Candidate
Paolo Montesel
University of Padova
Dept. of Information Engineering

February 21, 2017

Academic Year 2016/2017

To my parents and my brothers
To the friends who supported me in these years

*" The only truly secure system is one that is powered off,
cast in a block of concrete and sealed in a lead-lined room
with armed guards – and even then I have my doubts."*

– Gene Spafford

An extract of this thesis is included in a research paper that we plan to publish at a major research venue in the computer security area.

Abstract

Smartphones use has spread in many aspects of the everyday life of billions of users all across the globe. The vast amount of personal data stored on mobile devices is extremely sensitive in that smartphones have become a digital extension of their users' identities. Unfortunately, current systems do not allow the user to be fully in control of his device and his data. Mobile devices pose a significant challenge to security researchers as they need to be both secure and easy to use for the average user. For this reason, it is crucial to implement a security model that empowers the user to control his data and the way in which Apps manipulate and share it.

In this thesis we propose *UserLoop*, a system which makes use of sensor data and human feedback to enforce user awareness of sensitive actions performed by mobile applications. We present the architecture of the proposed system, describe our prototype implementation and report on the thorough evaluation we run in order to assess its performance both with regards to its effectiveness and its runtime footprint. The result of our evaluation on the prototype system confirms the feasibility of the system and its ability to prevent permission abuses performed by malicious Apps. Moreover, it shows that the overhead introduced by *UserLoop* doesn't impair the functionalities of the device.

Contents

Abstract **i**

1 Introduction **1**
1.1 Motivation . . . 2
1.2 Contribution . . . 3
1.3 Organization . . . 3

2 State of the Art **5**
2.1 Android source code modification . . . 6
2.2 Firmware Modification . . . 8
2.3 Application Repackaging . . . 9
2.4 Sandbox-based approaches . . . 10

3 Android **11**
3.1 Security model . . . 12
3.2 Android Permission Check . . . 13

4 Our proposal: UserLoop **15**
4.1 Architecture . . . 16
4.2 XUserLoop . . . 17
4.3 Context Tracker . . . 17
4.4 Policy Manager . . . 18
4.5 Client App . . . 19

5 Implementation **23**
5.1 Xposed . . . 23
5.2 XUserLoop . . . 24
5.2.1 *UserLoop* Service Injection . . . 24
5.2.2 Policy Enforcement Point . . . 27
5.3 ContextTracker . . . 29
5.3.1 Listening for Activity focus changes . . . 29
5.3.2 Detecting Input Events . . . 30
5.3.3 Tracking the screen lock state . . . 31

5.3.4 Tracking the screen state . . . 31
5.4 Logging . . . 32

6 Evaluation **33**
6.1 Security . . . 33
6.2 Performance . . . 36

7 Conclusion **43**
7.1 Future work . . . 43

Appendix **47**

A Configuration File **47**

B *UserLoop* Interface Definition **53**

List of Figures

2.1 Main Android Security Extensions [9] 5
2.2 *CRePE* Architecture [3] 7
2.3 Overview of *MOSES* [14] 7
2.4 *SAINT* Overview [17] 8
2.5 *FlaskDroid* Architecture [5] 9

3.1 Android Resource Access Framework [19] 12
3.2 Android permission history [13] 13

4.1 *UserLoop* Architecture 16
4.2 Detailed views of an App inside the GUI 20
4.3 *UserLoop* GUI 21

6.1 Policies violated per App 37
6.2 Battery level over three hours 38
6.3 Median overhead with increasing number of policies (values in ms) 41

List of Tables

6.1 Policy set 34
6.2 Permissions blocked 36
6.3 Mean operation duration with varying number of policies (results in ms) . 40

Chapter 1

Introduction

In recent years, smartphones have become the preferred way to access the Internet [1], and the use of Apps is now pervasive in many aspects of people's lives. The security and privacy of such devices is challenging due to the amount of personal data stored on them, the ever increasing presence of sensors which can be abused and the constant Internet connection.

The security model of mobile OSes is slowly improving but it is still far from ideal and the ease-of-use seems to often be favored when it comes to trade-offs between the two. Android versions prior to 6.0 implemented a permission system which required the user to accept all the permissions required by an App at install time without a way to revoke them selectively. This all-or-nothing approach was limited in that the user had no knowledge of why a permission was requested nor when it was requested. Android 6.0 has seen the introduction of Runtime Permissions, a feature which allows to toggle permissions on and off after the installation of an App. This has been a significant improvement, but it is not enough to protect users' privacy [2]. The main issue is that the model is still based on the concept of a set of coarse-grained permissions granted by the user to an App. After being granted a permission, Android does not stop a malicious App from using it to perform actions without user consent.

In order to tackle the limitations of the status quo, security researchers have proposed a wide range of security models and extensions. In fact, the open source nature of the Android OS has facilitated the study of its security model and therefore a significant body of literature is dedicated to it. There have been some focus on context-aware models which also consider information such as location, time and other sensors while performing Access Control at run-time [3, 4, 5]. These systems are based on predefined policies which must be created by the end-user or a third-party and therefore require a high degree of security expertise of the policy maker to be effective. For this reason, they are best suited for enterprise environments where security administrators are in charge of defining policies. Moreover, none of these proposals aims at using contextual information to provide a truly user-centered security system. As a result, their ability to make sure that the user is in the loop is limited.

Another key challenge in securing smartphones is the presence of many constraints that must be kept in mind during the design phase. For example, special care must be

taken when implementing security features which could significantly increase the power consumption of a device. Responsiveness and battery life are two of the most important factor to consider and failing to do so results in a degradation of the user experience. It must also be noted that, while the power of smartphones' CPU is steadily increasing, battery capacity have not followed Moore's Law [6] and so battery life remains the single most problematic component of a system.

1.1 Motivation

Recent history has shown that the users' privacy must not only be protected against malware, but also against benign Applications. In fact, cases have been reported of widely used Apps requiring too many permissions or leaking user information to their servers intentionally [7]. To make things even worse, several models of cheap Android handsets have been found to be transmitting personally identifiable information, including call logs and the content of text messages, to remote servers [8].

Despite the introduction of several extensions to the Android security model, little attention has been paid to possible mechanisms able to check whether a sensitive action is intended by the user or not, based on contextual information. This shortcoming severely limits the ability of a system to actually be secure and protect its users. For example, let us consider the case of a geo-tagging photo App. Such an App would typically require camera and location information for legitimate purposes. Moreover, Internet access might be needed to share the pictures among its users. In this scenario, nothing prevents the App from leaking the user location continuously while running in background or while the screen is off.

The permission granting process could be improved by using sensors' data and other contextual information, thus making it a well-informed task. Examples of such information could be knowing whether the screen is on/off, the time since last user input, the presence of the user in front of the screen or the rotation/position of the phone itself. Permission enforcement systems should exploit this data to try to block privileged but unsolicited actions requested by an App, therefore limiting potential privacy breaches.

In this thesis we introduce *UserLoop*, a user-centric and interaction-aware Access Control model which is, to the best of our knowledge, the first attempt of its kind to try to improve defenses against privacy leaks. *UserLoop* aims at making sure that the user is in the loop when permissions are requested and, consequently, it aims at blocking actions which exploit permissions granted by the user but used in an unintended way. To make the system effective, we provide the user with a reasonable set of sane default policies which he is able to edit in an intuitive way through a simple client App. In order to make this process easier, we also provide per-App statistics in a graphical manner to guide his decisions. Feedback from the user is essential for the system to be effective as it's difficult to cover all the edge cases a priori. Still, the defaults can be instrumental in reducing the user frustration in defining common scenarios.

1.2 Contribution

In this thesis we propose, to the best of our knowledge, the first user-centric and context-based policy enforcement security model that uses data from sensors to enforce user-awareness of sensitive actions: *UserLoop*. We present the full architecture of our system and describe its implementation as an extension of the Android security model. We also prove that, since we build upon Android's built-in security model, the security of our system is at least as good as Android's.

The concept of context-based access control has already been investigated on mobile devices by the research community, but no system so far has exploited contextual information to put the user and its interaction with the device in the center of the security model. Moreover, existing systems have a limited definition of context which is mostly static and is used only to change the set of policies depending on environmental attributes such as the time of the day or the location of the device. *UserLoop*, on the other hand, can apply different policies based on the instantaneous value of any of its contextual variables, thus adapting automatically to the user's needs. Furthermore, the *UserLoop* client App provides an easy and effective way for the user to detect suspicious Apps which perform sensitive requests behind his back.

With *UserLoop*, we achieve a fine-grained control policy over Android Apps by making the default Android permission model context-dependent. Our system doesn't require installing custom ROMs or radical changes to the OS, thus giving it the ability to be deployed on different devices and different Android versions with minimal to no changes. *UserLoop*'s only requirement is root access at install time.

The measurements performed on our prototype implementation show that *UserLoop*'s approach is feasible and provide a baseline for research into similar user-centric security models. Furthermore, the overhead of the prototype is negligible and does not impair the user-experience.

The security evaluation we performed proves that *UserLoop* is effective in preventing privacy leaks while the user is not in-the-loop. It also shows that even mainstream Apps perform several sensitive operations without user consent.

Finally, our system also poses as a framework for future analysis of Apps' behavior when the user is not in-the-loop. In fact, by storing the permissions requested in a central database along with their respective contextual information, it opens up the possibility of using machine-learning techniques to improve the security of the user.

1.3 Organization

We hereby describe the general structure of this thesis. In chapter 2, we discuss the state of the art of security research on the Android OS, laying out the main security extensions with their strengths and their weaknesses. We also analyze the approaches available when it comes to implementing such extensions. In chapter 3, we briefly describe the Android OS and its execution model. We then turn to describing its key security

features, with special attention to its history and the current trends. Furthermore, we discuss its limitations and possible improvements. Chapter 4 introduces the architecture of *UserLoop* and gives and overview of its various modules and their purpose. In chapter 5, we delve into the implementation details of our system, discussing system API hooking and context tracking. Chapter 6 presents the methodology we used to evaluate *UserLoop* and its prototype implementation. Moreover, it discusses the results of the performance and security measurements we performed on our test device. Finally, in chapter 7 we draw our conclusions and explore possible directions for future research using or extending *UserLoop*.

Chapter 2

State of the Art

The subject of improving the security architecture of Android, and smartphones in general, has been widely researched and still is. Mainstream research has given much attention to the topic of improving Access Control in Android devices by creating several extensions to the built-in features [9][10][4][3][11][5][12]. Some of the main Android security extensions are shown in Figure 2.1. In this chapter we analyze the main approaches used to implement security extensions in mobile devices.

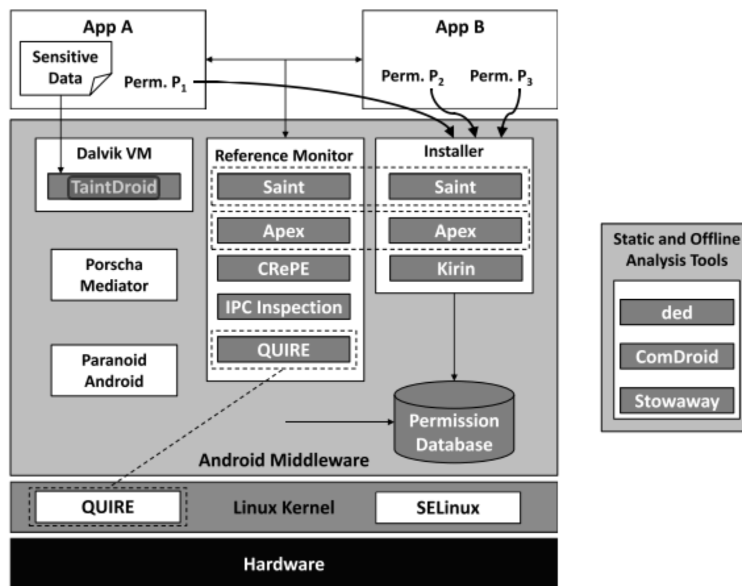

Figure 2.1: Main Android Security Extensions [9]

The rest of the chapter describes several security extensions proposed in the literature. The systems are grouped into sections according to the underlying technique used to implement them.

In section 2.1, we discuss the systems using the most straightforward method for implementing security extensions: modifying the Android source code. This approach is

the most flexible but comes with several drawbacks.

Section 2.2 describes firmware modification-based systems and their advantage in comparison with the ones using source code modification.

Section 2.3 briefly introduces the concept of application repackaging, a technique used to implement security extensions without modifying the operating system.

Finally, section 2.4 describes two novel security architectures which employ a sandbox-based approach to implement radically new security models.

2.1 Android source code modification

Android's source code is openly available and this makes it easy to implement security extensions by modifying the Operating System directly. Doing so allows researchers to alter significantly the existing system and is the most flexible option among the one analyzed in this thesis.

The Android ecosystem is getting mature, however its permission model does not show signs of moving towards more fine-grained permissions. In fact, around 44.8% of the Apps have been found to request permissions that they do not need, thus violating the least-privilege principle [13]. This, combined with lack of flexibility in the security model configuration, is what lead researchers to propose several Android security extensions.

In [3] the authors describe *CRePE*, a context-aware system capable of enforcing fine-grained policies in Android phones. Figure 2.2 shows the architecture of the system. As in our work, their system intercepts the Android permission check at the framework level. The context is defined as a set of variables such as position, time, light, etc. This approach is interesting but makes a limited use of sensors, requires the user (or a third-party) to manually define a set of rules and doesn't take into account the interaction of the user with the device.

The automatic use of context tracking to dynamically switch security profiles has been investigated in *MOSES* [14]. This work introduces the concept of "mode of use" and security profiles to enforce different policies for accessing applications and data. Switching profiles doesn't require user intervention but the system still requires the user, or a third-party, to provide the profile definitions. Figure 2.3 shows how *MOSES* implements different security profiles related to, for example, the working and the private environments.

Apex [4] allows the user to selectively grant permissions to Apps as well as impose constraints on their use of resources. It also considered the end-user side of the security equation by providing an easy-to-use interface to set such constraints from the package installer. Since Android 6.0, the ability to selectively deny or grant permissions has been integrated in the system by Google.

COMPAC [15] turns Android's UID-based security model into a UID and component-based one. It allows users and developers to assign a subset of an application's permissions to some of its components. In particular, *COMPAC* allows to group Java packages into components, which in turn are what the Apps are made of. Using this hierarchical separation of an App into smaller units, *COMPAC* can enforce different policies for different parts of

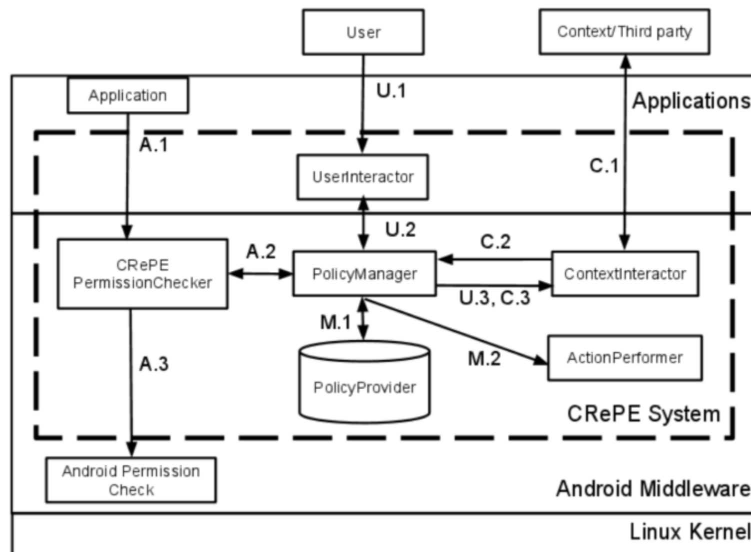

Figure 2.2: *CRePE* Architecture [3]

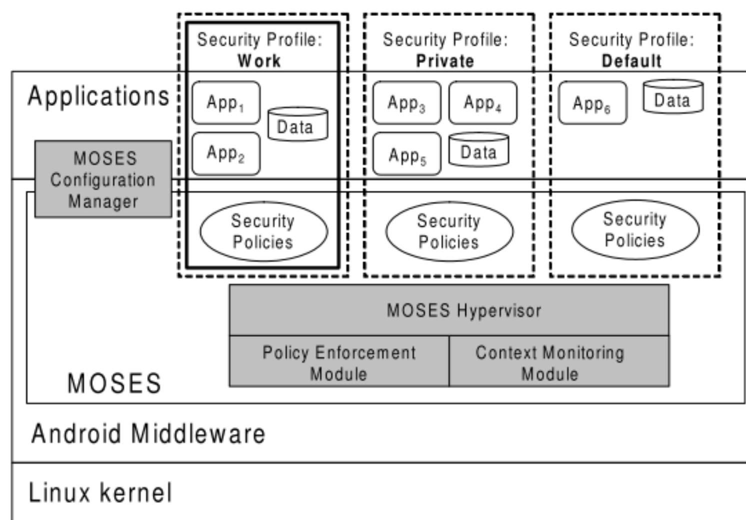

Figure 2.3: Overview of *MOSES* [14]

a single App. This approach can be used to restrict the access to sensitive permissions and enforce the least-privilege principle on Android.

In *TISSA* [16], the authors introduce the concept of *privacy mode* for Android devices. Such functionality allows users to control in a fine-grained manner the sensitive information that is made accessible to an App.

SAINT [17] takes on the diametrically opposite issue of creating a framework to allow App developers to defend from other Apps. The authors note that Android provide limited functionalities for developers to specify how the interfaces and the resources of their applications are accessed by third-parties. As such, *SAINT* gives Apps the ability to

provide install-time policies that regulate the access to their interfaces. *SAINT* policies can make use of contextual information to dynamically change the restrictions at runtime. Figure 2.4 shows the overview of the *SAINT* system. As said above, the policy file is shipped together with the APK.

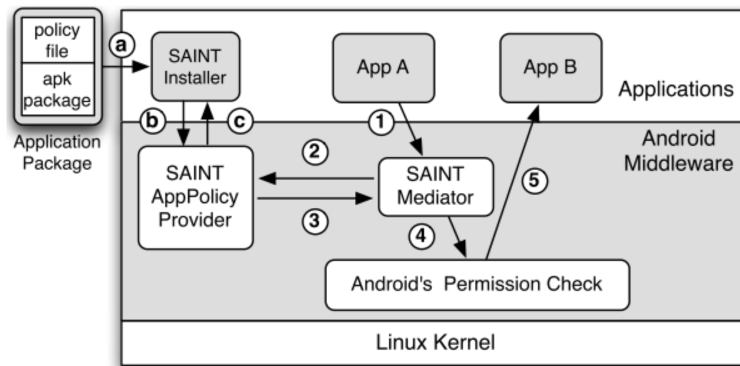

Figure 2.4: *SAINT* Overview [17]

An interesting approach to Android Access Control has been studied in [5]. The authors introduce a novel security architecture, named *FlaskDroid*, which uses a SELinux-inspired policy language and works at both the framework and the kernel level. *FlaskDroid* leverages *SELinux* to dynamically enforce policies through the use of boolean flags that are toggled at runtime. This gives the ability to implement context-dependent policies and have them enforced at all the levels of the Android software stack. The authors make use of contextual information in the policy engine but it's limited to environmental variables that don't capture user interaction and awareness. Still, *FlaskDroid* is interesting in that, ruling out Linux kernel exploits, it is able to enforce policies even if system process are compromised. This is because *SELinux* is able to perform mandatory access control on both the kernel and the framework level.

In order to reduce the need for pre-defined policies, [18] introduces a context-aware and adaptive security framework for Android which makes use of sensors data to inform a user about whether a requested action poses a low or a high security risk. They model the risk assessment as a Multi-criteria Decision Making problem and use Analytic Hierarchy Process to deal with it. This system could be used to bypass the need for user-defined policies, but the authors don't consider user awareness in their model.

2.2 Firmware Modification

A less invasive technique when it comes to implementing security extensions to the Android OS is firmware modification. This technique involves modifying some files on the system in order to extend the capability of the OS without flashing a complete ROM. The two main drawbacks are that root privilege is required and that heavily-customized ROMs may not

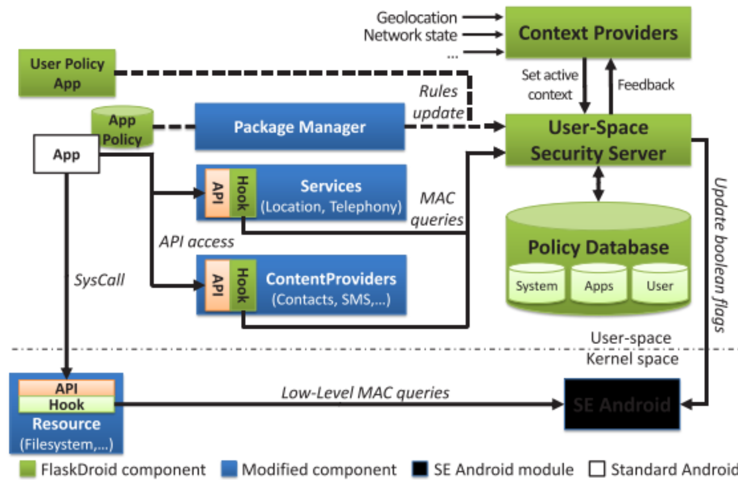

Figure 2.5: *FlaskDroid* Architecture [5]

work well together with the extension. Still, this approach usually works on a wide range of devices without requiring changes to the source code and is therefore an interesting method.

DeepDroid [19], a dynamic enterprise security policy enforcement scheme, allows for the definition of fine-grained context-based policies. Their approach is based on the dynamic instrumentation of both Dalvik and native code. In addition to the ordinary information about the permissions requested by an App, it can extract extra information (e.g., an URL associated with an http request). Being targeted at enterprise, *DeepDroid* requires a pre-defined set of fine-grained policies to be effective. Moreover, it fails to consider the user interaction and to exploit user feedback.

Considering the lack of source code for most commercial devices, we chose to implement *UserLoop* with this technique.

2.3 Application Repackaging

Application repackaging is a technique which bypasses completely the need to modify the Android source code or the firmware installed on a device. Instead of extending the security model of the operating system, repackaging works by modifying the Apps before installing them. Hooks and policy enforcement logic are injected into the App binaries and therefore systems using this technique do not need to adapt the Android OS to their needs. The main drawback of repackaging systems is that the security guarantees are only as good as the tool that analyses and repackages the Apps. If a malicious App is obfuscated well enough, it could prevent the hooks from being inserted into its code, thus making the system useless. *Aurasium* [10] is an example of this technique.

2.4 Sandbox-based approaches

Boxify [20] leverages Android's isolated processes to run applications in a sandbox. A key advantage of this work is that it doesn't require any system nor application modification as it is implemented as a normal Android application. As most systems it suffers from the lack of kernel support for policy enforcement, as that would require custom kernel extensions. Moreover, the authors themselves acknowledge that *Boxify* requires a wide set of permissions to be able to provide them to the sandboxed Apps. This could be exploited by malicious Apps due to the fact that Android doesn't provide any mean to drop permissions at runtime. While being an interesting general purpose Access Control mechanism, the authors don't provide information about applications of their system.

Another novel approach is introduced by [21] with the *FlowFence* system. *FlowFence* performs the equivalent of taint analysis by making data and control flows of the Apps explicit. This system is targeted at IoT security, but its core concepts could also be applied to the Android OS. *FlowFence* is implemented by making use of Android's isolated processes. Such processes have restricted IPC capabilities and permissions and are therefore a perfect candidate to implement sandboxes. In order to make the flows explicit, *FlowFence* separates Apps into Quarantined Modules and normal code. The formers are small snippets of code which operate on sensitive data inside a sandbox. In the system, sensitive data cannot escape the sandbox and normal code only sees an *opaque handle* when the data is returned from a sandbox. Apps derive their functionality by chaining together Quarantined Modules and declaring their flows, which must be approved either by the information sources/sinks or by the final user. This approach is quite interesting but is completely different from the permission-based approach of Android an thus can't be easily integrated with it.

Chapter 3

Android

Android is an open-source, Linux-based mobile operating system developed by Google and the Open Handset Alliance. Applications are written in a mix of Java and C/C++, with the former being the most popular and officially recommended option. Once compiled, the Java Virtual Machine (JVM) bytecode is translated to a custom format that is run by the Dalvik Virtual Machine (DVM). Android 4.4 saw the introduction of the Android Runtime (ART), an experimental new runtime environment that became the default since version 5.0 [22]. It introduced ahead-of-time compilation taking Dalvik binaries as input and outputting ELF binaries.

In Android, applications are started by cloning an initial process, named *Zygote*, which is afterwards given proper permissions and Linux user identifier (UID). Such UID is unique and assigned to an App at install time and guarantees that each application runs in a separate process, isolated from the others. Inter-Process Communication (IPC) is performed through a well-defined mechanism implemented and enforced at the framework level: the *Binder*. In the Linux kernel, the *Binder* is implemented as a custom driver and that is capable of serializing and unserializing complex objects to transfer them across process boundaries.

In section 3.1, we lay out how the Android security model works and what are its main features. We then explain shortly how it has improved over the years and what is still lacking from a user-centric security point of view. The final part deals with the worrying trend of the number of new dangerous permissions being introduced with every Android release.

Finally, in section 3.2, we describe how the Android permission check is performed when an App makes a request. In particular, we note that since Android 6.0, the procedure has become a bit cumbersome because of the need to maintain retro-compatibility with Apps written for older versions of the OS. This poses a security threat due to the fact that, even on recent Android versions, Apps targeting older versions of the SDK are still checked with the old procedure. Thus, they don't require explicit user consent when requesting dangerous permissions at runtime.

3.1 Security model

Security wise, Android uses a multi-layer security model with a mix of both Discretionary Access Control (DAC) and Mandatory Access Control (MAC). It implements a kernel-level application sandbox by leveraging UIDs and UNIX-style file permissions. Since version 4.3 of the OS, *SELinux* was introduced and from version 4.4 it started being deployed in enforcing mode. At the framework level, MAC is enforced through the use of labels called permissions. Such permissions, once granted to an App, allow it to perform privileged operations and to access protected features of the OS (e.g., Internet connectivity). Apps who want to make use of them must declare in their manifest file the set of permissions that they will use in their lifetime. Moreover, since Android 6.0, sensitive permissions must be explicitly requested to the user at runtime and they can potentially be selectively disabled afterwards. Figure 3.1 shows an abstracted overview of the Android resource access framework and the permission checks it performs.

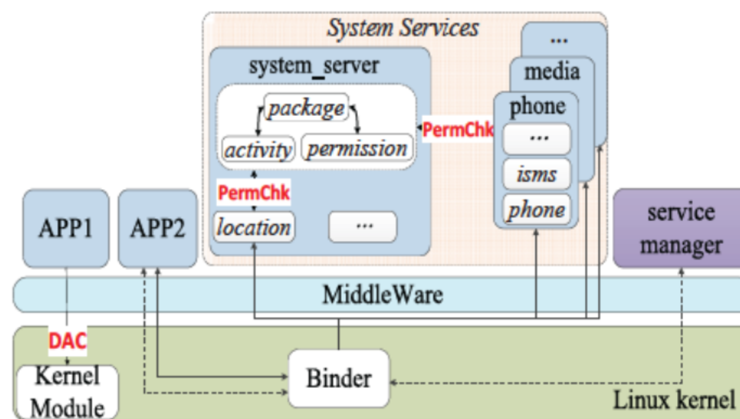

Figure 3.1: Android Resource Access Framework [19]

Beside the introduction of runtime permissions, the security of Android has recently seen other improvements on the user side. In version 5.0, Google introduced Smart Lock, a system which allows the user to unlock the phone using a trusted device, such as a Bluetooth/NFC beacon, a smart-watch or through facial recognition. In version 6.0, the company also introduced a fingerprint API. All these features are a step forward in making security easier for the average user and do so by exploiting some contextual information. For example, a Bluetooth beacon provides an effective way to check whether a legitimate user is close to his device or not. An improved permission model based on these principles could successfully reduce privacy breaches while keeping the friction for the end user at a minimum.

However, Android's security model is still based on a set of coarse-grained permissions. With every new version of the OS the permission set has been extended but the permissions themselves have not been made more fine-grained [13]. What's more alarming is that the subset of *dangerous* permissions is the largest one and that the subset of *signature* or

signatureOrSystem permissions has been expanded significantly, as shown in Figure 3.2. Permissions labeled as *signature* are only available to Apps signed with the same certificate as the application that declared the permission. Similarly, *signatureOrSystem* has the same policy but grants access also for applications shipped with the Android system image. It must be noted that such permissions are granted automatically if the requirements are met, thus bypassing completely the end-user.

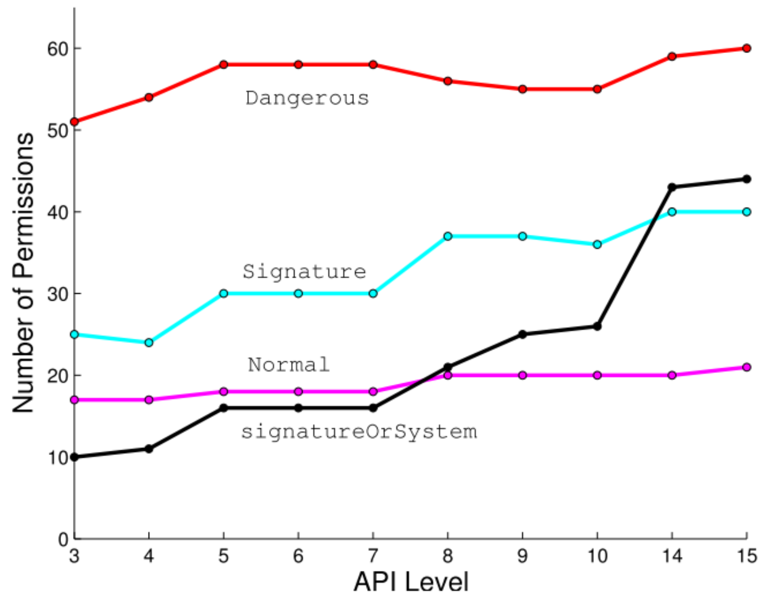

Figure 3.2: Android permission history [13]

3.2 Android Permission Check

In Android versions before 6.0, App permissions were granted at install time without the ability to disable them selectively. Since 6.0, dangerous permissions are now asked explicitly to the user when requested the first time and then granted automatically. Due to retro-compatibility concerns, Apps targeting SDKs before version 23 are still subjected to the old behavior even on modern devices. The high-level flowchart of how permissions are granted, depending on the SDK version, is shown in Figure 3.3.

Whenever an App requests a privileged operation, such request is serialized and sent to the *system_server*, the process which hosts most of the default Android services. Here, it reaches the service in charge of processing it. Before actually performing any sensitive operation, every service calls the `checkUidPermission(String perm, int uid)` method of the *PackageManager* service. Thus, this method is where the actual permission check is performed, using just the permission name and the UID of the requesting App. In *UserLoop* we append custom logic at the end of `checkUidPermission` to implement additional security checks.

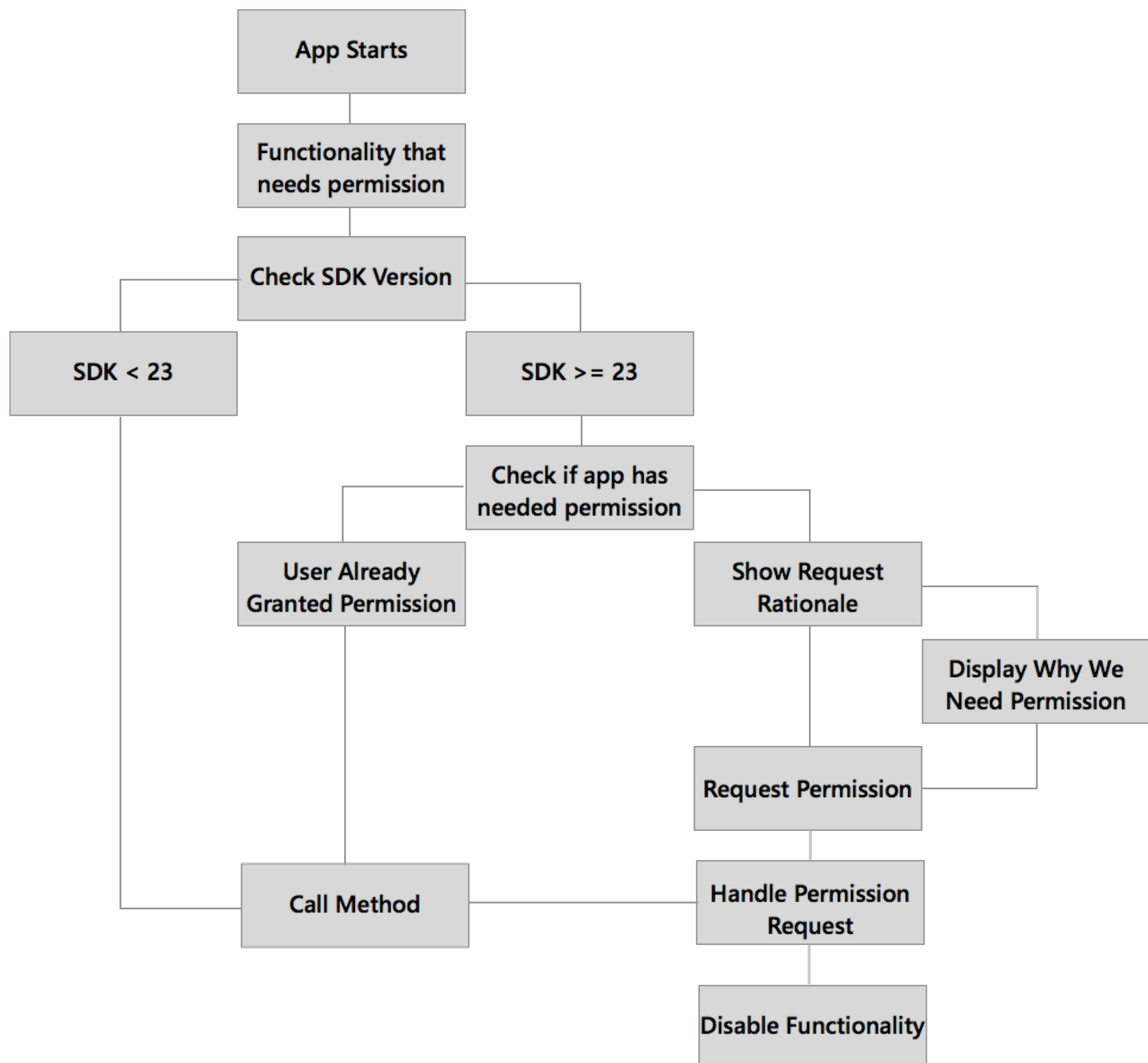

Figure 3.3: Android permission work-flow [23]

Chapter 4

Our proposal: UserLoop

This chapter discusses the high-level architecture of the *UserLoop* security model. *UserLoop* is designed in a modular fashion in order to decouple its parts as much as possible. Moreover, it is split between the client App in user-space and the core system which is injected at runtime into the Android middleware. This provide us with a high degree of flexibility when it comes to changing the system or implementing new features.

The first section, 4.1, introduces the *UserLoop* architecture with special focus on the interplay between the different modules of the system. It also lays out the basic structure of the permission checking routine and show how it is positioned with respect to Android's built-in permission check.

Then, section 4.2 briefly discusses the role of the *XUserLoop* module in starting *UserLoop*, registering it as a system service and performing the actual permission checks using the information gathered from the other modules.

After that, in section 4.3, we show how *UserLoop* aggregates different sensors and contextual variables in order to achieve the goal of estimating whether the user is in-the-loop or not. We also analyze how every sensor could give *UserLoop* information about the state of the interaction with the user.

In section 4.4, we discuss how the *UserLoop* policies are defined from the logical point of view. The section also deals with how our system deals with multiple intersecting policies in order to avoid conflicts. Moreover, the role of the default policy and permission grouping is explained.

Finally, section 4.5 introduces the client App and the way it interacts with the *UserLoop* service. We discuss how having a GUI is essential in order to allow the end-user to change and review the policies deployed by default. The section also showcases a simple scenario that gives an idea of how, just by using the client App, a user could be able to spot malicious Apps and their permission abuses.

4.1 Architecture

The architecture of *UserLoop* is shown in Figure 4.1. *UserLoop* is composed of three main modules: an *Xposed* module named *XUserLoop*, a service and a client App. *XUserLoop* is the Policy Enforcement Point (PEP) and is used to relay Android's UID-based permission check to the *UserLoop* service. The service is itself composed of different modules which encapsulate its main functionalities. The *Logger* module is in charge of storing permission requests together with contextual information in order to provide useful statistics to the front-end App. The *Context Tracker* is the module which gathers data from the sensors and keeps track of the current context. The *Policy Manager* acquires contextual information from the *Context Tracker* and provides *XUserLoop* with policies to enforce. It also communicates with the *Policy Provider*, which is where the policies are stored. Finally, the client App, the user-facing part of the system, is used to manipulate policies and show the user useful information on the Apps and their permission requests. The client can also update the *UserLoop* configuration whenever the user make a change to a policy.

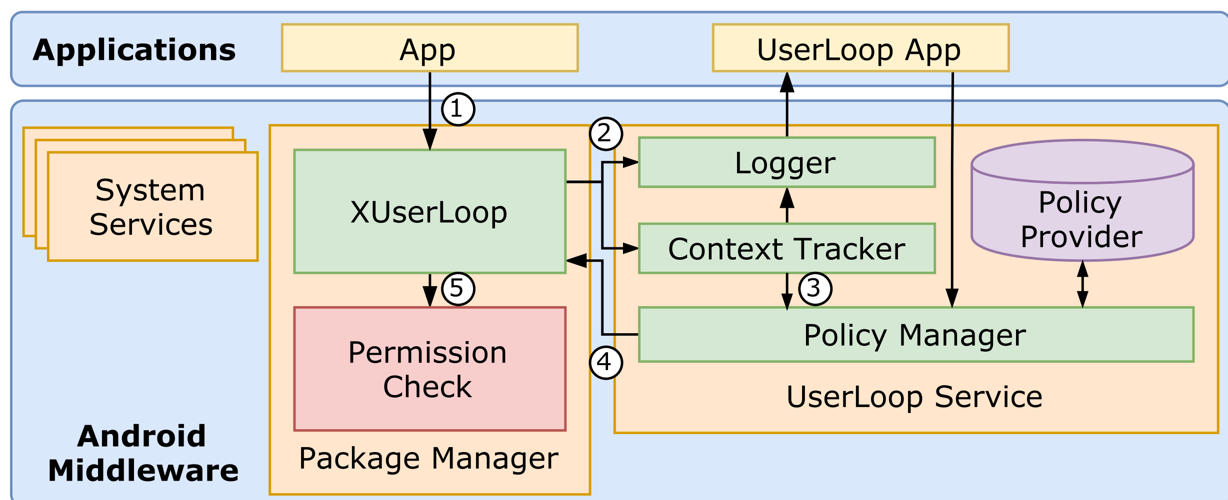

Figure 4.1: *UserLoop* Architecture

The numbered arrows in the diagram follow the flow of a permission request inside the system. Whenever an App requests a privileged action through the Android Framework, a permission check is performed inside the `PackageManager.checkUidPermission(String permName, int uid)` method ①. The string `permName` uniquely identifies the permission being requested, while `uid` is the Linux UID of the requesting App. If the check performed by Android has a positive outcome, *XUserLoop* intercepts the return value and performs additional checks by forwarding the request to the *UserLoop* service ②. Here the request is logged in a database and handed to the *Context Tracker*, which appends contextual information and hands it to the *Policy Manager* ③. At this point, the system goes through the policies to determine if one of them applies and the result is sent back to the *Xposed* module ④. The final step is the actual enforcement of the policy ⑤ which can result in one

of the following scenarios:

1. No policy applies.

2. At least one policy applies.

In the first case, the system applies the default policy. Such policy is user-defined and could grant or deny the permission based on the configuration set by the user inside the *UserLoop* App. In the latter case, the first matching policy is applied. *UserLoop* also provides a whitelist and a blacklist that can be used to bypass the rule system in special cases. The entries to those lists are tuples composed of a package name and a set of permissions. The use of a whitelist is especially useful in that it can be used to grant vital permissions to important services like messaging Apps. At the same time, the blacklist can be used to manually deny specific permission to Apps that the user does not trust.

4.2 XUserLoop

XUserLoop is the module in charge of placing the hooks into the Android OS. As the name suggests, it relies heavily on the *Xposed* framework to achieve this goal.

XUserLoop intercepts the Android booting process and, at the right moment, registers the *UserLoop* sevice, thus making it available to the client App. It also gets a reference to the *system_server* `Context` instance, as it needed for most operations involving the Android framework.

By hooking the `checkUidPermission(String permName, int uid)` method, *XUserLoop* becomes the PEP of our security model. Whenever the Android `PackageManager` grants a permission to an App, this module intercepts the return value of the check in order to let the *UserLoop* service perform its additional checks. If Android's permission checking routine denies a request, *XUserLoop* simply returns the original value as we are only interested in making the Android security model stricter. Thus, *UserLoop* never grants a request that was denied by Android.

The explanation of *XUserLoop*'s inner workings is laid out in chapter 5 as it mostly involve the technical details of hooking the internal components of the Android OS.

4.3 Context Tracker

The Context Tracker is one of the main modules of our system and provides contextual information to the rest of *UserLoop*. In *UserLoop*, the context consists of a mix of data gathered from the sensors and of information about software-based states. This module is one of the key elements of our system, as every policy enforcement performed by *UserLoop* uses the information about the context to grant or deny a permission request. Thus, modeling the user context in an effective way is crucial in making the user more secure.

Being interested in understanding whether the user is using the phone or not, the most valuable sensors are the rotation vector sensor, the accelerometer, the light sensor and

the proximity sensor. The rotation vector and the accelerometer are used to estimate the position of the device in the 3D space and also to estimate the level of physical engagement of the user. For example, if the accelerometer reports that the phone is in a still position it is very likely that the user is not actually using it, even if the screen is on. The proximity sensor is instrumental in checking if the phone is upside-down on a table, in a pocket or if the user is facing the screen. Finally, the light sensor can augment the proximity data and can also be used to check whether the user is sleeping or not.

In addition to these common Android sensors, *UserLoop* makes use of other information from the OS. We identify five values which can significantly improve the value of *UserLoop*'s security model: the screen state, the lock state, the visibility state of the App, the process state and the time since the last user input. The first two are simple on/off boolean values and are used to represent, respectively, if the screen is on or if it is locked. The visibility state of the App is also a boolean value which encodes whether the App is in foreground or in the background when it makes a permission request. The process state takes a value of either `system` or `user` and enables us to differentiate between system Apps and user-installed Apps. Finally, the time since the last user action is recorded. This is useful to recognize scenarios in which the screen is on but the user is not paying attention and is not actually interacting with the device. For this variable, we keep track of both the global time since the last input and the time since the last input inside the App making the permission request.

4.4 Policy Manager

Whenever an App requires a permission, the request reaches the *Policy Manager* module with contextual information attached to it. Policies are defined as an ordered list of tuples

$$(permissionType, permission, dangerous, conditions, action),$$

where

- *permissionType* $\in \{SINGLE, GROUP, NULL\}$ defines if *permission* refers to a permission or to a permission group.
- *permission* $\in$ *Permissions* $\cup$ *PermissionGroups* $\cup \{NULL\}$ is the name of a permission or a permission group.
- *dangerous* $\in \{TRUE, FALSE, NULL\}$ defines if the rule applies to dangerous permissions, normal permissions or both.
- *conditions* is a list of conditions composed of a *UserLoop* contextual variable, a comparison operator and a value. Such conditions are concatenated through the use of multiple AND boolean operators.
- *action* $\in \{GRANT, DENY\}$ defines the action that must be taken if all the conditions are satisfied.

In the policy definition, *NULL* attributes are considered to be matching regardless of their actual value. We note that the user is spared from dealing directly with the policy implementation language thanks to the client App.

UserLoop eliminates the risk of overlapping or conflicting policies by checking them in a cascading manner. Because of this, the closer the policy to the start of the policy list the higher its priority. *Policy Manager* also holds a global policy which has an empty condition and therefore defines the default outcome when no other policy applies. The final user can decide if this global policy should grant or deny requests not covered by any rule. The *Policy Manager* interacts with the *Policy Provider* to retrieve and store policies. Thus, the latter is basically a storage for policies.

In order to provide a more effective control on the permissions, they are grouped according to the Android permission grouping. Such grouping differs from an Android version to another and so groups are created at runtime by inspecting the `Manifest` class. Using groups allows, for example, to use of a single *UserLoop* policy for both coarse and fine location requests.

4.5 Client App

We provide a front end App to *UserLoop* in order to let the user review and tune the policies shipped by default. Moreover, since we gather statistics about the permission requests, we are able to show some useful information to let him quickly grasp if an App is suspicious. For this reason, the *UserLoop* client App contributes to our security model by giving feedback to the system.

In order to showcase the usefulness of *UserLoop* and its GUI, we developed a custom App named *EvilApp*. *EvilApp* is a simple parking App which asks for the location information to save it and later tell the user where he parked his car. We chose this simple scenario because the Play Store is flooded with free Apps providing similar functionalities. Once *EvilApp* gets the location permission, it sets up a periodic timer which is used to record the user position without him being aware of it. The timer is also restarted after every boot in order to make the threat persistent across reboots. We then deployed two simple policies to deny dangerous permissions when an App is in background or when it has not received user input recently. Figure 4.2 is an example of what the user would see upon clicking on *EvilApp* inside the *UserLoop* GUI. It is immediately clear that it violated the policies several times and also that it is recording the user location while it is running in background. Figure 4.3 shows four of the main views present inside the client App. In the current iteration of *UserLoop*, the data is stored on the device only for visualization purposes, but with minimal changes it could be sent to a remote server to perform more complex analysis.

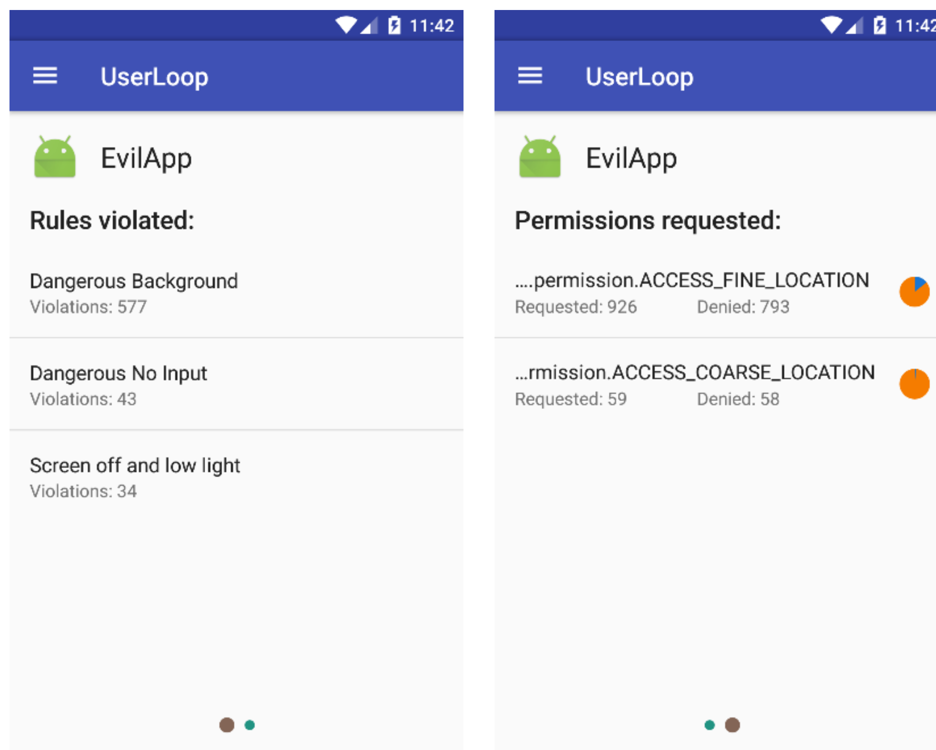

Figure 4.2: Detailed views of an App inside the GUI

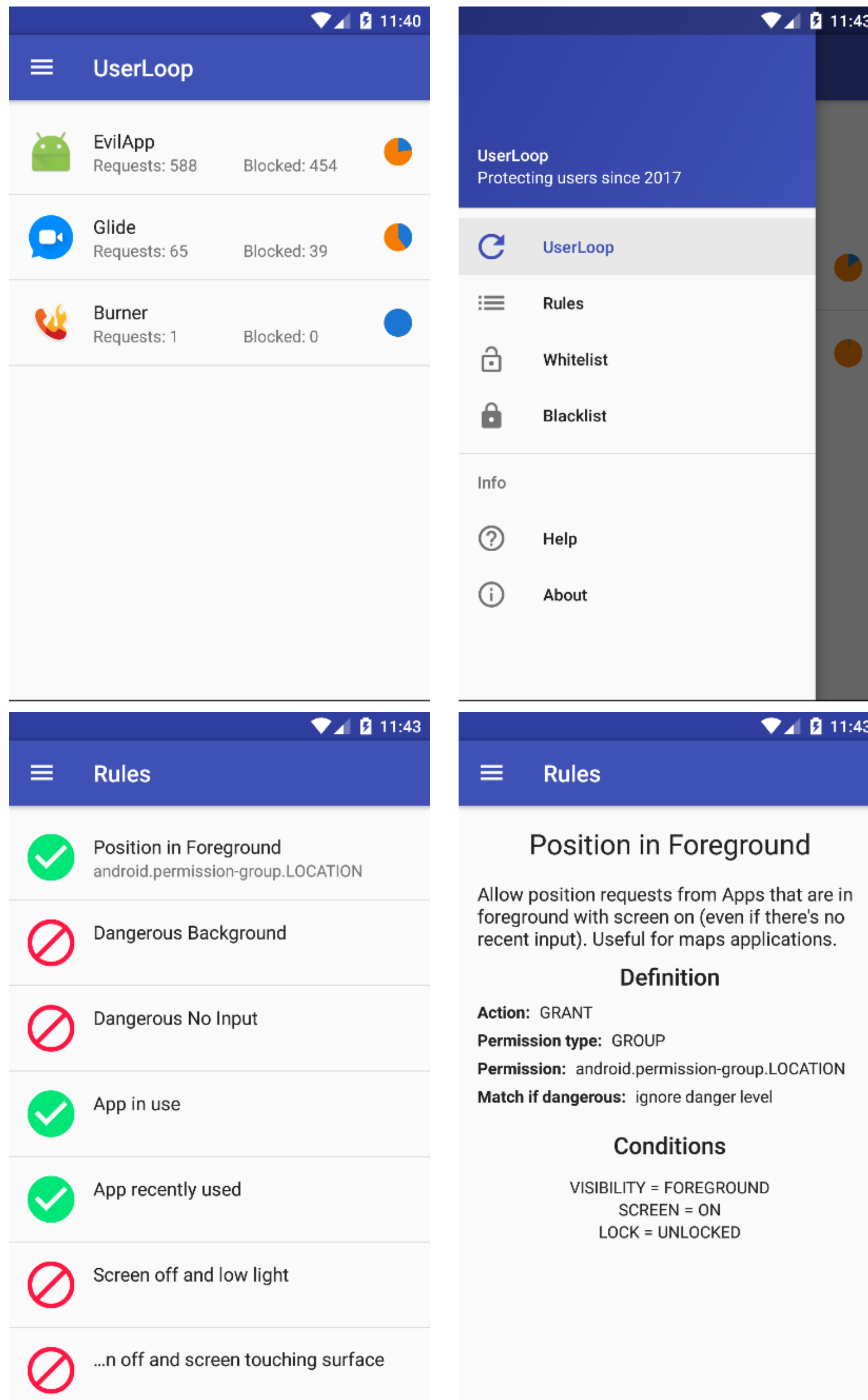

Figure 4.3: *UserLoop* GUI

Chapter 5

Implementation

This chapter discusses the implementation of the main components and the hooking logic used by *UserLoop*. Our system works at the framework level and is therefore implemented in Java.

We begin in section 5.1 by giving a brief introduction of *Xposed*, the framework used to implement the hooks needed by *XUserLoop*. We also analyze the choice of using it instead of modifying directly the Android source code.

We then discuss *XUserLoop* itself in section 5.2. The module is central in bootstrapping the *UserLoop* service at system startup and in placing the hooks required to perform the *UserLoop* permission check.

Section 5.3 deals with the challenges faced in implementing an effective and power efficient context tracking mechanism and how they were overcome.

Finally, section 5.4 gives an high-level view on how we implemented the permission request logging in *UserLoop*, with a focus in keeping the system responsive even under a load higher than normal.

5.1 Xposed

Xposed is a framework that makes it easy to change the behavior of any App on an Android device without modifying any APK. It achieves this goal by allowing third-party modules to hook java methods from other packages. Despite being an unofficial runtime modification tool, it has seen wide adoption due to its flexibility and its ability to modify system Apps without requiring the users to flash a custom ROM. For this reason, we chose to use it to implement our system hooking routines.

Xposed works by installing a custom `app_process` binary that is run at system startup instead of the original one. The extended binary, once loaded, injects an additional jar file in the *classpath*. This happens before the `main` of *Zygote* is called and therefore gives the ability to perform operations inside the *Zygote* process itself[24].

The power of the framework comes from the fact that if provides facilities to hook and replace any java method from any package on the system. This includes system Apps and

system service and so it can be used to hook the system `Context`. In *UserLoop* we exploit this functionality to register a custom service inside *system_server*. This is essentially the same as modifying the Android source code and installing a custom ROM, but without having to actually do it.

The main drawback of using *Xposed* is that it requires root privileges at installation time. Moreover, they must be granted after every system update as updates wipe it from the system by restoring the stock `app_process`. Still, *Xposed* doesn't technically require root privileges after it has been installed.

From the security point of view, *Xposed* could pose a threat as malicious Apps could trick the user into installing modules which would compromise the entire system. In fact, while *Xposed* modules do not have root privileges, they can inject custom code in every Android process and thus have control over potentially every part of the system. For this reason, the *Xposed* module manager requires the user to manually activate a module after it has been installed by an App.

5.2 XUserLoop

In this section we discuss *XUserLoop*, the component that performs all of the Android OS hookings, starts the *UserLoop* service and execute our custom permission check using contextual information.

5.2.1 *UserLoop* Service Injection

One of the key challenges in implementing *UserLoop* was to find the proper places in the Android framework where to place the hooks needed for it to work. The first step that we need to do in order to get the server part of our system up and running is to hook into the Android startup procedure. To properly inject our custom service inside *system_server*, we need to wait long enough for the system to perform the initial setup of its core services, but not too long. In fact, if *UserLoop* starts too late, its security could be compromised because a malicious App would be able to bypass it. After evaluating different options, we decided to place our hooking logic at the end of the `systemMain()` method of `ActivityThread`. From here we are able to place hooks to get a reference to both the system `Context` and the `WindowManagerService`, which will be used later to intercept touch-screen input before it's delivered to the Apps. The actual startup of *UserLoop* itself is delayed until the `systemReady()` method of `ActivityManagerService` is invoked by Android.

The initial hooking is performed with the following code, executed inside *Zygote*:

```
1 Class<?> at = Class.forName("android.app.ActivityThread");
2 XposedBridge.hookAllMethods(at, "systemMain", new XC_MethodHook()
    {
3   @Override
4   protected void afterHookedMethod(MethodHookParam param) throws
      Throwable {
```

```
5        ... //Code to be executed at system startup
6     }
7  }
```

From inside the hook above, we get a reference to the system instance of `Context` by intercepting the constructor of `ActivityManagerService`:

```
1  final ClassLoader loader = Thread.currentThread().
      getContextClassLoader();
2
3  // Hook the system context
4  final Class<?> am = Class.forName(
5    "com.android.server.am.ActivityManagerService", false, loader
6  );
7  XposedBridge.hookAllConstructors(am, new XC_MethodHook() {
8    @Override
9    protected void afterHookedMethod(MethodHookParam param) throws
        Throwable {
10     mSystemContext = (Context) param.args[0];
11   }
12 });
```

On line 10, *UserLoop* gets the reference to the system context. Every system service in Android gets a copy of it as an argument to its constructor. In fact, just as for normal Android Apps, most of the operations performed on the device require access to a `Context` instance.

UserLoop then gets a reference to the `WindowManagerService` in a similar way:

```
1  // Get an instance of WindowManagerService
2  final Class<?> wm = Class.forName(
3    "com.android.server.wm.WindowManagerService", false, loader
4  );
5  XposedBridge.hookAllConstructors(wm, new XC_MethodHook() {
6    @Override
7    protected void afterHookedMethod(MethodHookParam param) throws
        Throwable {
8      mWindowManagerService = param.thisObject;
9    }
10 });
```

By hooking the constructor of the `WindowManagerService`, *UserLoop* is able to get access to the instance just created. In *Xposed*, such instance is contained inside the `thisObject` field of the hook argument `param`, of type `MethodHookParam`.

At this point, *UserLoop* has just to wait for the call to `systemReady()` and register its service before the method is executed. The operation just described is performed by placing an additional hook inside of the `ActivityManagerService`:

```java
// Register the UserLoop service when the system is up and running
XposedBridge.hookAllMethods(
  am,
  "systemReady",
  new XC_MethodHook() {
    @Override
    protected final void beforeHookedMethod(final MethodHookParam
       param) {
      if (mSystemContext == null) {
        d("System UserLoopContext is null :/");
      }

      UserLoopService.register(
        mSystemContext, mWindowManagerService
      );

      // Clean up unused references
      mSystemContext = null;
      mWindowManagerService = null;
    }
  }
);
```

Finally, the registration is performed by calling the method `addService(String serviceName, IBinder service, boolean allowIsolated)` of the Android `ServiceManager`. The first argument is the name of the service being added, which will later be used to access the functionalities it offers. The second argument is the instance of the service that must be added. Its type is `IBinder` and this means that the service must provide a *Binder* interface. This is usually done by creating an *Android Interface Definition Language* (AIDL) file and letting the SDK tools generate the needed `Stub` interface. The complete *UserLoop* interface definition is listed in Appendix B. The last argument is a boolean flag which indicates whether or not Android's isolated processes can access the service. In the current iteration of the implementation it is set to `true`, but in future versions it will be disabled as it shouldn't be needed.

Unfortunately, the `ServiceManager` is not available in the standard SDK as it cannot be used by regular Apps, so *UserLoop* needs to access it through the Java reflection API:

```java
static void register(Context systemContext, Object wm) {
  d("Registering...");

  mUserLoopService = new UserLoopService(systemContext, wm);

  Class<?> cServiceManager = null;
  try {
    d("Adding service...");
```

```
9       cServiceManager = Class.forName("android.os.ServiceManager",
           false, null);
10      Method mAddService = cServiceManager.getDeclaredMethod("
           addService", String.class, IBinder.class,
11      boolean.class);
12      mAddService.invoke(null, SERVICE_NAME, mUserLoopService, true)
           ;
13
14      mUserLoopService.init();
15      d("Service added succesfully!");
16    } catch (ClassNotFoundException e) {
17      e.printStackTrace();
18    } catch (NoSuchMethodException e) {
19      e.printStackTrace();
20    } catch (IllegalAccessException e) {
21      e.printStackTrace();
22    } catch (InvocationTargetException e) {
23      e.printStackTrace();
24    }
25  }
```

On line 4, the *UserLoop* service instance is created and saved in a static variable. On line 12, the service is added to the *system_server*.

5.2.2 Policy Enforcement Point

As explained in chapter 4, every privileged request in the Android framework reaches the `checkUidPermission(String permName, int uid)` method of class `PackageManager` at some point. This is because such method is where Android's permission checking is performed. It is therefore the perfect place where to add the additional permissiong checking routine that *UserLoop* must execute.

In order to add our custom login inside the `PackageManager`, we exploit *Xposed*'s ability to set a listener that triggers when a new package is loaded by the Android OS:

```
1  @Override
2  public void handleLoadPackage(XC_LoadPackage.LoadPackageParam lpp)
       throws Throwable {
3    if ("android".equals(lpp.packageName)) {
4      d("Hooking system_server...");
```

Here, `handleLoadPackage(...)` gets called every time a package is loaded. Since *UserLoop* is only interested in the `PackageManager`, it does not do anything if the hook is being executed in the wrong process. `PackageManager` is one of the many services hosted by inside the *system_server*, whose process is named `android`.

If the check is successful, *UserLoop* proceeds by placing its permission checking logic after `checkUidPermission(...)`:

```
5     XposedHelpers.findAndHookMethod(
6       "com.android.server.pm.PackageManagerService",
7       lpp.classLoader,
8       "checkUidPermission",
9       String.class,
10      "int",
11      new XC_MethodHook() {
12      @Override
13      protected void afterHookedMethod(MethodHookParam param)
           throws Throwable {
```

When the hook is executed, our custom logic is run inside `PackageManager`. Since the hooks are placed before *UserLoop* is actually started, we first need to check if it is running:

```
14        // If system is still being initialized
15        if (!UserLoopService.isRegistered()) {
16          return;
17        }
```

If it is not running yet, we confirm the decision taken by `PackageManager` and return immediately. It must be noted that the permission requests made before *UserLoop* is started are performed by the Android OS, so allowing them *no matter what* does not pose a security threat.

UserLoop then checks if the permission was already denied by the `PackageManager`. In such case, *UserLoop* doesn't need to perform additional checks since it only provides stricter security guarantees than stock Android:

```
18        // If Android denied it we don't need to do anything else
19        boolean androidDenied = ((int) param.getResult()) ==
             PackageManager.PERMISSION_DENIED;
20        if (androidDenied) {
21          return;
22        }
```

Finally, *XUserLoop* reaches the *UserLoop* PEP and thus forward the permission check to the `UserLoopService`:

```
23        // If we reach this point, it means that Android pre-
             approved the request and we
24        // now need to enforce UserLoop policies.
25        PermissionAction action = UserLoopService.getClient().
             checkUidPermission(
26          (String) param.args[0], (int) param.args[1]
27        );
28        param.setResult(action.equals(PermissionAction.GRANT) ?
             PackageManager.PERMISSION_GRANTED : PackageManager.
             PERMISSION_DENIED);
```

```
29         }
30       });
31     }
32 }
```

5.3 ContextTracker

Context Tracker is the *UserLoop* module that keeps track of the user context and hands this information to the permission checking routine when needed. Most of the values that compose the context are gathered from the sensors on the device with the usual APIs that Android provides for user-installed Apps. For example, data from sensors such as the accelerometer and the proximity sensor is recorded through a standard listener registered in Android's `SensorManager`. There are, however, four contextual variables that need special handling: keeping track of the Activity currently in foreground, detecting input events, checking whether the device is locked or not and checking if the screen is on.

In the following subsections we explain how we implemented the aforementioned operations.

5.3.1 Listening for Activity focus changes

The definition of *UserLoop* context requires knowledge about which App the user is currently interacting with. In other words, *UserLoop* needs to always know the identity of the App in foreground.

One way to achieve this goal is by polling the Android API that gives information about the running tasks. However, polling is slow and not very precise: if an Activity is started between a call to the API and the next, it would be treated as if it where in background. Moreover, polling requires continuous *busy-waiting*, a situation that is better to avoid in a resource-constrained device such as a smartphone.

After evaluating different options, we settled on using *Xposed* once again to hook into the `ActivityManagerService` method `setFocusedActivityLocked(ActivityRecord r, String reason)`. This method is called by the Android framework whenever an Activity needs to be in foreground. The `ActivityRecord` argument contains informations about the App the Activity belongs to, including its UID.

Using a hook instead of a polling mechanism has the advantage of avoiding the busy-waiting that is inevitable with the latter method. The code *UserLoop* uses to implement this behavior is given in the following snippet:

```
1 XposedBridge.hookAllMethods(
2   am,
3   "setFocusedActivityLocked",
4   new XC_MethodHook() {
5     @Override
```

```
6      protected void afterHookedMethod(MethodHookParam param) throws
         Throwable {
7        if (param.args[0] == null) {
8          return;
9        }
10
11       // param = { ActivityRecord, String reason }
12       Class<?> cActivityRecord = Class.forName(
13         "com.android.server.am.ActivityRecord", false, loader
14       );
15       Field fAppInfo = cActivityRecord.getDeclaredField("appInfo")
           ;
16       fAppInfo.setAccessible(true);
17       ApplicationInfo appInfo = (ApplicationInfo) fAppInfo.get(
           param.args[0]);
18       UserLoopService.getClient().setFocusedApp(appInfo.uid);
19     }
20   }
21 );
```

Once the UID is retrieved and given to the *UserLoop* service (line 18), it is available to the *Context Tracker* at any moment.

5.3.2 Detecting Input Events

UserLoop's context contains two variables which record the time since last input: one refers to touches local to the App requesting a permission, the second to the global input of the device. Android performs its low-level input handling routines with native code and there is not an API to intercept all user inputs, probably for legitimate security concerns.

The Android OS has an internal class, `WindowManagerPolicy`, which allows for other parts of the framework to register pointer event listeners. This seems to be the best place to get information about all the touchscreen inputs, but unfortunately this functionality is not available in the standard SDK. For this reason, we used Java's reflection API to call the `registerPointerEventListener(...)` method:

```
1 final Class<?> cPointerEventListener = Class.forName(
2   "android.view.WindowManagerPolicy$PointerEventListener"
3 );
4 Method mRegisterPtrListener = mWindowManager.getClass().
     getDeclaredMethod(
5   "registerPointerEventListener", cPointerEventListener
6 );
7 Object oPointerEventListener = Proxy.newProxyInstance(
8   cPointerEventListener.getClassLoader(),
9   new Class[]{cPointerEventListener},
```

```
10   new InvocationHandler() {
11     @Override
12     public Object invoke(Object proxy, Method method, Object[]
          args) throws Throwable {
13       if ("equals".equals(method.getName())) {
14         if (!cPointerEventListener.isInstance(proxy)) {
15           Log.d(TAG, "not instance");
16           return false;
17         } else {
18           return proxy == args[0];
19         }
20       } else if ("onPointerEvent".equals(method.getName())) {
21         onMotionEvent((MotionEvent) args[0]);
22         return null;
23       }
24       Log.d(TAG, "Proxy method not implemented: " + method.getName
            ());
25       return null;
26     }
27   }
28 );
29 mRegisterPtrListener.invoke(mWindowManager, oPointerEventListener)
     ;
```

On line 7, a new instance of `PointerEventListener` is created and is then registered in the last line of the snippet.

5.3.3 Tracking the screen lock state

In order to give *UserLoop* the ability to behave differently when the screen of the device is locked, it needs access to such information whenever a permission request reaches the PEP.

Fortunately, Android provide the state of the screen lock by a simple call to a standard SDK API. This API is part of the *Keyguard* service and is implemented in *UserLoop* in the following way:

```
1 public boolean isLocked() {
2   KeyguardManager mKeyguardManager = (KeyguardManager)
       mSystemContext.getSystemService(Context.KEYGUARD_SERVICE);
3   return mKeyguardManager.inKeyguardRestrictedInputMode();
4 }
```

5.3.4 Tracking the screen state

Knowing the screen state is another useful piece of information when implementing a context-based security model such as *UserLoop*.

As for the screen lock, Android provides an official API to query the state of the screen. Such API is managed by the *DisplayManager* service, which also supports multi-screen devices. Being targeted at common smartphones, *UserLoop* considers the screens as being on if at least one of them is on. In normal devices, this reduces to checking if the only screen present is on. The following snippet of code shows the current implementation of the functionality just described:

```
1 public boolean isScreenOn() {
2     DisplayManager dm = (DisplayManager) mSystemContext.
          getSystemService(Context.DISPLAY_SERVICE);
3     for (Display display : dm.getDisplays()) {
4       if (display.getState() != Display.STATE_OFF) {
5         return true;
6       }
7     }
8     return false;
9 }
```

5.4 Logging

The key functionality that is needed to provide useful information to the user in the *UserLoop* client App is logging the permission requests. The *Logger* module stores not only the identity of the App requesting a permission, but also the complete content of the *UserLoop* context at the moment of the request.

Instead of using a textual log, we opted for using a SQLite database. The database has a single table, named `log`, which stores a permission request per row. The main advantages of this approach are that SQLite support is already baked in in Android and that it allows the GUI to perform complex SQL queries, needed to visualize the data to the user, in a clean way.

The client App implements a `ContentProvider` on top of the database to give the *UserLoop* service the ability to insert new rows through the standard Android IPC mechanisms. The service performs the row insertion asynchronously in a separate thread in order to avoid blocking the *system_server* when a high number of permissions are being requested in a short time.

Chapter 6

Evaluation

In this chapter we discuss the thorough evaluation of our system by assessing its security performance and its overhead compared with a clean device.

In particular, section 6.1 discusses the increase of the security offered by the Android OS when equipped with *UserLoop*. We evaluated our system on a selection of free Apps from the Play Store. Our results show that performing privileged operations while the user is not in-the-loop in common practice among the Apps we analyzed.

In section 6.2 we evaluate the performance overhead introduced by *UserLoop*. We first profiled the smartphone during normal usage in order to get a list of the privacy-related permissions requested most frequently. For each permission, we then chose an operation that triggers it and then performed a micro-benchmark with and without *UserLoop*. We also measured how the overhead scales with the number of policies deployed on the system.

Our test device is a Nexus 6 from *Motorola Mobility*, running stock Android 6.0 and *Xposed* version 87.

6.1 Security

We evaluated the security performance of our implementation using a sample of free Apps from the Play Store. The default policy chosen for testing is to deny every request, but a policy was added to grant non dangerous permissions to balance usability and security. Moreover, system Apps are assumed to be safe and so they are trusted regardless of the contextual information. Finally, we note that messaging Apps require constant internet connection and contact list access and must therefore be whitelisted accordingly. Other edge cases such as GPS navigation Apps, if they are used in background, must also be whitelisted.

Before delving into empirical results, we observe that *UserLoop* comes into play only after Android has already granted a permission to an App. For this reason, we are able to conclude that it does not reduce the Android security. In fact, by performing additional checks its security is always at least as good as Android's. In our analysis we don't consider privacy breaches that happen while the user is interacting with the device as leaks, as those

would be prevented effectively only by not using an App.

For the evaluation, we chose 6 free Apps: *Facebook*, *Skype*, *ES File Explorer*, *360 Security - Antivirus*, *Text Free* by *Pinger, Inc* and *Glide*. The first four are widely known and have more than 1M downloads while the others are less known but still have over 100.000 downloads. *Facebook* and *Skype* were chosen because, even if they come from well-respected companies, their history raises privacy concerns. *ES File Explorer* is a famous file manager which has recently started showing intrusive ads and including adware, according to many user comments on the Google Play Store[1]. *360 Security - Antivirus* is a free antivirus which claims to also optimize background processes, memory space and battery power. We chose to include it in our tests as an example of a ”boosting” app, because such Apps often come with intrusive ads and nagging notifications[2]. The last two Apps, *Text Free* and *Glide*, were reported to send potentially sensitive data to both its servers and to third-parties [25].

For the assessment of our system, we deployed the set of policies defined in Table 6.1. During our preliminary testing we found that accelerometer and gyroscope data wasn't very useful with regards to our aim of making sure the user is in the loop. This is because the current iteration of our system uses the instant value from the sensors, neglecting the time dimension. Further work is required to see if an activity recognition-based approach would yield better results.

Table 6.1: Policy set

ID	Condition	Action
R0	Location when App in foreground, screen on, phone unlocked	GRANT
R1	Dangerous permission, App in background	DENY
R2	Dangerous permission, no touch in the last 20 seconds	DENY
R3	App in foreground, screen on, phone unlocked	GRANT
R4	Screen on and unlocked, recent input inside the App	GRANT
R5	Screen off, low light	DENY
R6	Screen off, low proximity	DENY
R7	SEND_SMS permission, screen off	DENY
R8	CALL_PHONE permission, screen off	DENY
R9	Non-dangerous permission, screen on, phone unlocked	GRANT

In the initial phase of the security evaluation, we downloaded and installed the Apps on our test device. We then proceeded by manually opening them and creating accounts where required. After that, we performed manual testing by opening each App and interacting with it. In this phase, we modified some contextual values un purpose in order to activate the policies we deployed. Finally, we rebooted the phone and recorded the activity of the Apps for two days.

[1] `https://play.google.com/store/apps/details?id=com.estrongs.android.pop&hl=en`

[2] `https://www.reddit.com/r/androidapps/comments/4jqni5/list_of_apps_which_you_should_not_use_at_all/`

The complete list of permission requests that *UserLoop* blocked is shown in Table 6.2. We note that during our test *360 Security - Antivirus* crashed several times as a result of its permission requests being denied. We also note that the App makes an abnormal amount of requests for permissions while the user is not in the loop and this reveals its potentially malicious nature. As we expected, the permission blocked most frequently is ACCESS_NETWORK_STATE. This is because such permission is often used to check whether there is an active network connection or not, before actually conntecting to the Internet. While the network state itself isn't a very privacy-sensitive information, it allows Apps to potentially track the user through the SSID of the Wi-Fi networks the phone is connected to. For this reason, the decision to permit out-of-the-loop access to this information should be made on a per-App basis through the use of the whitelist. Another interesting result is that, with the exception of *Skype*, all the Apps request the GET_TASKS permission several times. This permission is used to get a list of the tasks running or recently visited by the user. Google deprecated it in Android 5.0 because of potential privacy leaks derived from the introduction of document-centric recents [26]. For example, Chrome tabs, if shown in the App switcher, could leak information about the pages recently visited by the user. To mitigate this threat, Android now returns a limited subset of information which is considered to be not sensitive. Our analysis shows that, despite being deprecated, GET_TASKS is still widely used. Moreover, it is requested when the user is not in the loop and is therefore blocked by *UserLoop*. The results for *Glide* and *Text Free* confirm previous findings [25]: they both request permissions several times while the user is not in the loop. The permissions, if granted, would give access to location, user accounts, contacts, phone number and network information. Location access is especially suspicious since these are messaging Apps and there doesn't seem to be a reason for them requesting such data so often. *ES File Explorer* requested the least amount of permissions but still tried to access bluetooth and show an overlay window when the user wasn't actively using the App. *Facebook* requested READ_CONTACTS and this suggest that it might transmit the data to its servers. While this could be used to suggest friends, we believe this to be a privacy leak anyway as the user is not explicitly informed of the action. *Skype* asks for the network state several times, but what is more interesting is its use of CALL_PHONE and WRITE_CONTACTS. The first permission is used to initiate a call without going through the Dialer App. In our test, the permission was requested a suspicious number of times even though no call was started. The second is also requested a non-negligible number of times.

Figure 6.1 shows which policies trigger *UserLoop*'s request denials. Our findings suggest that all the Apps perform sensitive operations while the user is not in the loop. *UserLoop* successfully blocks them but a manual analysis of the Apps is required to see how they operate on the information gathered. Unfortunately, it would involve reverse-engineering as the source code is not available and is thus excluded from our research at this point.

It is important to note that a vast amount of Apps still targets pre-M Android versions and therefore only ask for permission at install time. This happens even if the device is running a more recent release in order to avoid backward compatibility issues. For security reasons, we let *UserLoop* work also on pre-M Apps. Of course, this is a two-sided coin:

Table 6.2: Permissions blocked

Permission	**360 Security**	**ES File Explorer**	**Facebook**	**Glide**	**Skype**	**TextFree**
ACCESS_COARSE_LOCATION	3	0	0	2605	0	489
ACCESS_FINE_LOCATION	6	0	0	2606	0	0
ACCESS_NETWORK_STATE	96064	130	8552	3366	13083	19908
ACCESS_WIFI_STATE	0	3	162	0	13084	0
BLUETOOTH	0	16	0	0	0	0
CALL_PHONE	0	0	0	0	13078	0
GET_ACCOUNTS	79765	0	1	4216	0	6208
GET_TASKS	83641	70	148	878	0	5480
READ_CALL_LOG	0	0	0	1	0	0
READ_CONTACTS	0	0	57	67	7	6
READ_PHONE_STATE	39566	116	0	451	6	485
SYSTEM_ALERT_WINDOW	0	2	0	1	0	0
WRITE_CONTACTS	0	0	0	67	6544	0
WRITE_SETTINGS	0	27	0	0	0	0
Total:	299045	364	8920	14258	45802	32576

on one hand, we can provide increased privacy to the user, on the other hand, old Apps are likely to crash because they expect permissions to be granted at install time. Usually, other systems work around this problem by granting the permission but providing fake data to the requesting App. While this would work, it would also significantly increase the complexity of the system as it would require hooks in every sensitive API to be implemented effectively. We therefore leave this as an open issue for future investigations as it is beyond the scope of this paper.

6.2 Performance

We measured the power consumption and the timing of the permission check before and after the deployment of *UserLoop* in order to estimate its overhead in a real-world setting. We also estimated how the system scales with the increase in number of policies installed on a device.

In order to evaluate the power consumption of our system, we used the *Power Monitor*

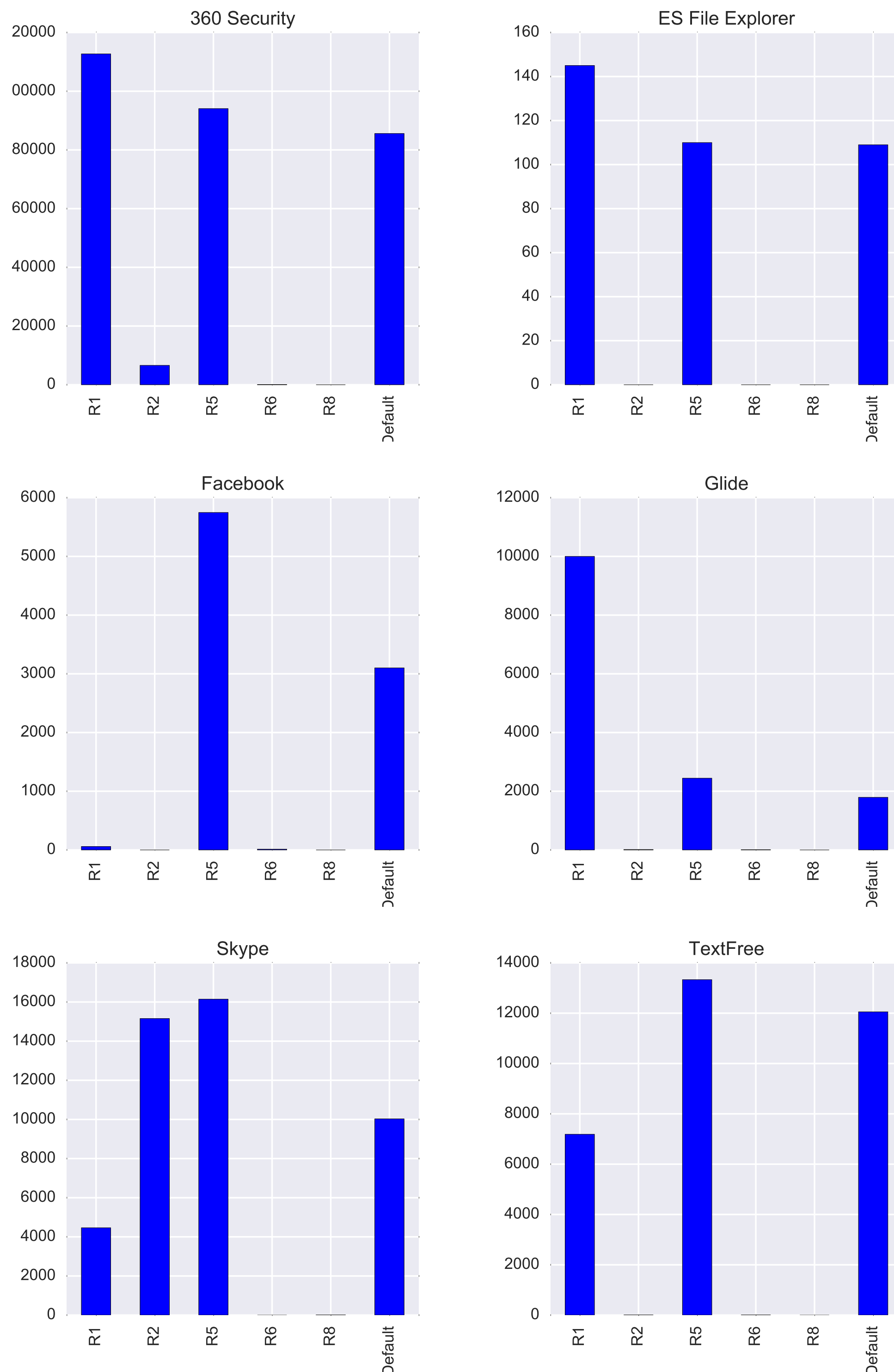

Figure 6.1: Policies violated per App

FTA22D, manufactured by *Monsoon Solutions, Inc*[3]. However, our measurements showed that in a small time-scale the increase in power consumption was undetectable. For this reason, we switched to measuring the battery drop over a longer period of time. We developed an App which requests a permission every second and let it run for three hours on a fully charged device. During a typical day usage, our device records an average of ≈1.38 permissions requests per second. System Apps account for ≈0.95 permissions per second while the rest comes from user-installed Apps. For this reason, by making a request every second we can simulate a scenario that is more demanding than what we recorded on our device. We then recorded the battery charge drop in terms of the percentage reported by the Android OS. By repeating this experiment, we are able to estimate how the energy consumption scales with the number of policies in place.

Figure 6.2 shows that the increase in power consumption induced by our system is within an acceptable range. In particular, during normal usage we estimate the number of policies to be well within 50. In such case, in a three hours period, the consumption is increased by only 1%. It should also be noted that, in a real world scenario, denying a permission for a computationally expensive operation could compensate for the increase in power consumption caused by *UserLoop*.

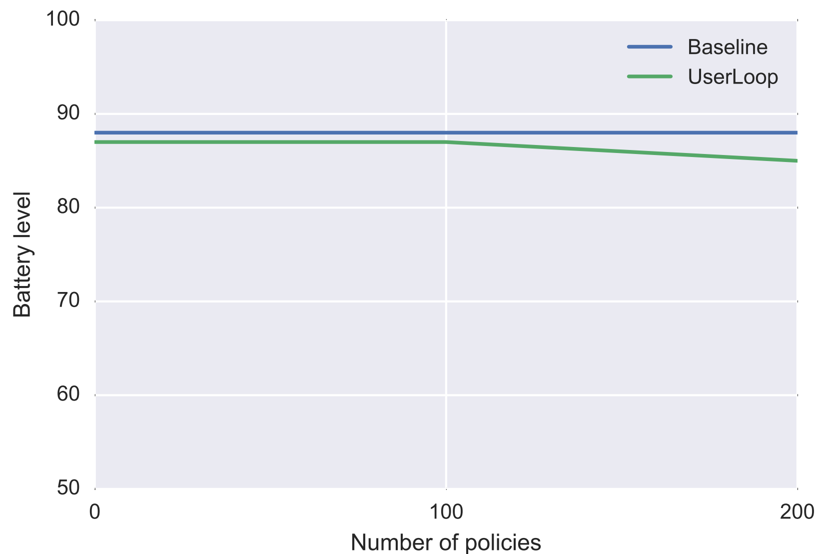

Figure 6.2: Battery level over three hours

We now proceed to evaluate the overhead introduced by *UserLoop* on the permission checking process. This evaluation is performed with a series of microbenchmarks on five operations which trigger permissions commonly requested by Apps. In order to choose the operations, we profiled our test system for an entire day and listed the most requested permissions. After removing system Apps and non-sensitive permissions we found

[3]`https://www.msoon.com/LabEquipment/PowerMonitor/`

that the most requested permissions or permission groups were, in order of frequency: READ_CONTACTS, GET_ACCOUNTS, READ_PHONE_STATE, WRITE_CONTACTS and LOCATION. We triggered them using the following operations:

- OP0: a query from Android's contact list
- OP1: *ActivityManager*'s *getAccounts()*
- OP2: *SubcriptionManager*'s *getActiveSubscriptionInfoForSimSlotIndex()*
- OP3: an update of a contact from Android's contact list
- OP4: *LocationManager*'s *getLastKnownLocation()*

We run each of these operations for 1000 times and measured the time required to get a return value.

Upon performing the first measurements, we noticed that the results weren't consistent. Our hypothesis was that the CPU frequency scaling governor was introducing an effect way stronger than the overhead produced by *UserLoop*. In fact, the default CPUFreq governor in the *Nexus 6* is *interactive*, a governor which is designed to decrease power consumption by adapting the CPU frequency to demand and user interaction [27]. We set a fixed CPU frequency equal to the one *interactive* sets immediately after a screen touch in order to reduce the variance introduced by the governor. After doing so, the timings were still subject to high variance but the results began being consistent across different runs. The results, per action, of the measurements are shown in Figure 6.3, where the central red line represents the median value of the timing samples. Table 6.3 shows the mean overhead measured by our benchmarks with varying number of policies deployed. The policies are randomly generated, target one of the aforementioned permissions and include the check for a contextual variable. We note that OP3 is a complex operation that requires the check for both the READ_CONTACTS and the WRITE_CONTACTS permission and the overhead is therefore higher than normal. The relative overhead measured by the benchmarks ranges from a minimum of $0.339ms$ to a maximum of $1.563ms$ (excluding OP3 results). For short operations, such as OP1, the overhead can be as high as 77%. For longer operations, such as OP0 and OP3, it is lower than 10%. We conclude that the performance penalty introduced by *UserLoop* is not negligible but, being well below $2ms$ in the worst case, it is reasonably within the threshold at which a human would notice it. We also stress that the current system implementation is not optimized for speed.

Table 6.3: Mean operation duration with varying number of policies (results in ms)

Operation	Baseline	0	10	20	30	40	50
OP0	14,70	15,70	16,01	15,46	15,62	16,27	15,49
OP1	0,77	1,11	1,15	1,21	1,24	1,31	1,37
OP2	3,69	4,92	5,12	5,09	5,04	5,17	5,17
OP3	49,00	52,16	52,35	52,57	51,38	52,95	52,72
OP4	1,10	1,78	1,77	1,84	1,81	1,79	1,83

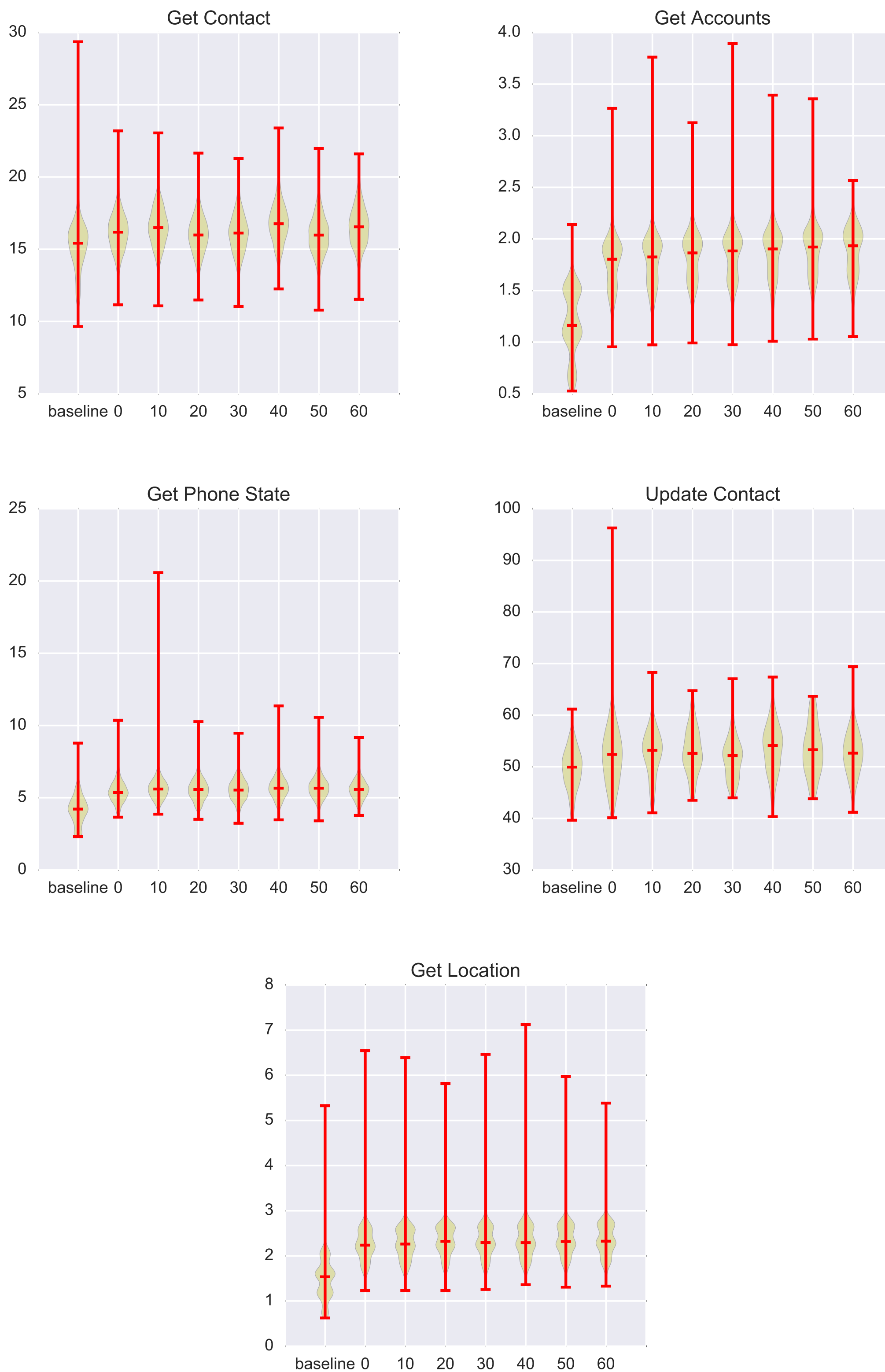

Figure 6.3: Median overhead with increasing number of policies (values in *ms*)

Chapter 7

Conclusion

In this work we presented *UserLoop*, a security extension to the Android OS that takes advantage of contextual information in order to block permissions requested when the user is not in the loop. *UserLoop* provides a mechanism to make the Android permission system more fine-grained through the use of context-dependent policies. Our evaluation shows that it is a viable way to give more control to the user over the Apps' usage of sensitive APIs. It also shows that Android applications abuse the permissions they are granted and use them when the user is not in the loop.

We believe that Apps should limit their usage of privacy-related operations while the user is not interacting with the phone and *UserLoop* proved to be effective in preventing this behavior. Moreover, by providing the user with information about permission requests and policies violated, it allows him to review the set of installed Apps. User awareness of privacy infringing applications could also result in negative reviews on the Play Store and therefore increase the security of the entire user base.

7.1 Future work

We acknowledge that the core implementation of the system hereby presented could be improved to make it more effective. To begin with, *UserLoop* works only at the framework level and so, for example, native applications could easily bypass it. It could be extended to perform policy enforcement also at the kernel level but this is beyond the scope of this thesis. In fact, as it is customary in security research, it is assumed that an adversary motivated enough can always find a way to bypass the implementation of a security model.

An interesting first step to improve *UserLoop* would be to analyze the data it records while it's operating. For example, data gathered from devices could be sent to a remote server in order to perform computationally-heavy tasks which are prohibitive even on a cutting-edge smartphone. This would allow us to apply machine learning techniques to mine policies or to create a system which automatically alerts the user if malicious activities are detected.

Machine learning could also be used to create a contextual model that, given raw sensor

data as the input, would create an abstracted set of variables in an activity-recognition fashion. It would be interesting to see how activity information could improve our model of the user context and therefore allow for more fine-tuned policies.

We also note that he current iteration of our work is not yet optimized for performance and our analysis shows that there is room for improvement. A faster permission check would allow us to reduce both the power consumption and the runtime overhead of *UserLoop*, thus making it more useful for the final user.

Finally, we plan to apply *UserLoop* to systematically evaluate Apps from the Play Store in order to get a better understanding of how and when permissions are requested. With this information we hope to be able to tune our system and make the end user more secure to privacy leaks.

Appendix

Appendix A

Configuration File

What follows is the configuration file used for the security evaluation. *UserLoop* configuration is stored in a `json` file named `config.json`:

```
1  // Comments are OK (Gson strips them out by default)
2  {
3    // We trust system apps
4    "trustSystemApps": true,
5    // Save some space
6    "logSystemApps": false,
7    "defaultAction": "DENY",
8
9    "blacklist": {
10     "com.android.camera2": [ "android.permission.INTERNET" ]
11   },
12   "whitelist": {
13     // WhatsApp
14     "com.whatsapp": [
15       "android.permission.ACCESS_NETWORK_STATE",
16       "android.permission.INTERNET"
17     ],
18     // Xposed Installer
19     "de.robv.android.xposed.installer": [
20       "android.permission.ACCESS_NETWORK_STATE",
21       "android.permission.INTERNET"
22     ],
23     // Facebook Messenger
24     "com.facebook.orca": [
25       "android.permission.ACCESS_NETWORK_STATE",
26       "android.permission.INTERNET"
27     ]
28   },
29   // Permissions we don't want to monitor
```

```
30    "ignorelist": [
31      "android.permission.CHANGE_WIFI_MULTICAST_STATE",
32      "android.permission.GET_PACKAGE_SIZE",
33      "android.permission.INTERNET",
34      "android.permission.READ_EXTERNAL_STORAGE",
35      "android.permission.WRITE_EXTERNAL_STORAGE",
36      "android.permission.VIBRATE",
37      "android.permission.WAKE_LOCK"
38    ],
39    "rules": [
40      {
41        "id": 0,
42        "name": "Position in Foreground",
43        "description": "Allow position requests from Apps that are
             in foreground with screen on (even if there's no recent
             input). Useful for maps applications.",
44        "permissionType": "GROUP",
45        "permission": "android.permission-group.LOCATION",
46        "dangerous": null,
47        "conditions": [
48          { "variable": "VISIBILITY", "operator": "EQ", "value": "
               FOREGROUND" },
49          { "variable": "SCREEN", "operator": "EQ", "value": "ON" },
50          { "variable": "LOCK", "operator": "EQ", "value": "UNLOCKED
               " }
51        ],
52        "action": "GRANT"
53      },
54      {
55        "id": 1,
56        "name": "Dangerous Background",
57        "description": "Block requests for dangerous permissions
             while an App is in background",
58        "permissionType": null,
59        "permission": null,
60        "dangerous": true,
61        "conditions": [
62          { "variable": "VISIBILITY", "operator": "EQ", "value": "
               BACKGROUND" }
63        ],
64        "action": "DENY"
65      },
66      {
67        "id": 2,
68        "name": "Dangerous No Input",
```

```
69        "description": "Block requests for dangerous permissions if
             the user hasn't touch the phone in a while",
70        "permissionType": null,
71        "permission": null,
72        "dangerous": true,
73        "conditions": [
74          { "variable": "GLOBAL_INPUT_AGE", "operator": "GT", "value
              ": 20 }
75        ],
76        "action": "DENY"
77      },
78      {
79        "id": 3,
80        "name": "App in use",
81        "description": "Allow requests from Apps in foreground",
82        "permissionType": null,
83        "permission": null,
84        "dangerous": null,
85        "conditions": [
86          { "variable": "SCREEN", "operator": "EQ", "value": "ON" },
87          { "variable": "LOCK", "operator": "EQ", "value": "UNLOCKED
              " },
88          { "variable": "VISIBILITY", "operator": "EQ", "value": "
              FOREGROUND" }
89        ],
90        "action": "GRANT"
91      },
92      {
93        "id": 4,
94        "name": "App recently used",
95        "description": "Allow requests from Apps with recent touch
             input",
96        "permissionType": null,
97        "permission": null,
98        "dangerous": null,
99        "conditions": [
100         { "variable": "SCREEN", "operator": "EQ", "value": "ON" },
101         { "variable": "LOCK", "operator": "EQ", "value": "UNLOCKED
              " },
102         { "variable": "APP_INPUT_AGE", "operator": "LT", "value":
              "6" }
103       ],
104       "action": "GRANT"
105     },
106     {
```

```
107        "id": 7,
108        "name": "Screen off and low light",
109        "description": "Likely sleeping or phone in the pocket",
110        "permissionType": null,
111        "permission": null,
112        "dangerous": null,
113        "conditions": [
114          { "variable": "SCREEN", "operator": "EQ", "value": "OFF"
                },
115          { "variable": "LIGHT", "operator": "LTEQ", "value": 10 }
116        ],
117        "action": "DENY"
118      },
119      {
120        "id": 8,
121        "name": "Screen off and screen touching surface",
122        "description": "Phone likely upside-down on a table or
               inside a bag/pocket",
123        "permissionType": null,
124        "permission": null,
125        "dangerous": null,
126        "conditions": [
127          { "variable": "SCREEN", "operator": "EQ", "value": "OFF"
                },
128          { "variable": "PROXIMITY", "operator": "LTEQ", "value": 1
                }
129        ],
130        "action": "DENY"
131      },
132      {
133        "id": 9,
134        "name": "SMS with screen off",
135        "description": "Don't send SMS when phone screen is off",
136        "permissionType": "SINGLE",
137        "permission": "android.permission.SEND_SMS",
138        "dangerous": null,
139        "conditions": [
140          { "variable": "SCREEN", "operator": "EQ", "value": "OFF" }
141        ],
142        "action": "DENY"
143      },
144      {
145        "id": 10,
146        "name": "Call phone with screen off",
147        "description": "Don't call when phone screen is off",
```

```
148        "permissionType": "SINGLE",
149        "permission": "android.permission.CALL_PHONE",
150        "dangerous": null,
151        "conditions": [
152          { "variable": "SCREEN", "operator": "EQ", "value": "OFF" }
153        ],
154        "action": "DENY"
155      },
156      {
157        "id": 9999,
158        "name": "Non dangerous",
159        "description": "Allow non-dangerous when the phone is in use
              ",
160        "permissionType": null,
161        "permission": null,
162        "dangerous": null,
163        "conditions": [
164          { "variable": "SCREEN", "operator": "EQ", "value": "ON" },
165          { "variable": "LOCK", "operator": "EQ", "value": "UNLOCKED
                " }
166        ],
167        "action": "GRANT"
168      }
169    ]
170  }
```

Appendix B

UserLoop Interface Definition

What follows is the definition of the *UserLoop* Binder interface as declared in the file `IUserLoopService.aidl`:

```
package it.unipd.montesel.userloop;

import it.unipd.montesel.userloop.config.PermissionAction;

interface IUserLoopService {
  // System Guarded
  void setFocusedApp(int userId);

  // UserLoop GUI Guarded
  void updateUidMap();
  void updateConfiguration(String config);

  // No Check
  PermissionAction checkUidPermission(String perm, int userId);
  String hello();
}
```

Bibliography

[1] The Guardian. *Smartphone now most popular way to browse internet.* 2015. URL: `https://www.theguardian.com/technology/2015/aug/06`.

[2] David Barrera et al. "A methodology for empirical analysis of permission-based security models and its application to android". In: *Proceedings of the 17th ACM conference on Computer and communications security - CCS '10*. 1. New York, New York, USA: ACM Press, 2010, p. 73. ISBN: 9781450302456.

[3] Mauro Conti, Vu Thien Nga Nguyen, and Bruno Crispo. "CRePE: Context-Related Policy Enforcement for Android". In: 2011, pp. 331–345. ISBN: 978-3-642-18178-8.

[4] Mohammad Nauman, Sohail Khan, and Xinwen Zhang. "Apex: Extending Android Permission Model and Enforcement with User-defined Runtime Constraints Mohammad". In: *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security - ASIACCS '10* (2010), p. 328. ISSN: 9781605589367.

[5] Sven Bugiel, Stephan Heuser, and Ahmad-Reza Sadeghi. "Flexible and fine-grained mandatory access control on Android for diverse security and privacy policies". In: *Proceedings of the 22nd USENIX Security Symposium* (2013), pp. 131–146.

[6] J.A. Paradiso and Thad Starner. "Energy Scavenging for Mobile and Wireless Electronics". In: *IEEE Pervasive Computing* 4.1 (2005), pp. 18–27. ISSN: 1536-1268.

[7] Arstechnica.com. *Your iPhone calendar isn't private.* 2012. URL: `http://arstechnica.com/apple/2012/06/your-iphone-c`.

[8] New York Times. *Secret Back Door in Some U.S. Phones Sent Data to China, Analysts Say.* 2016. URL: `http://www.nytimes.com/2016/11/16/us/politics/china-phones-software-security.html`.

[9] Sven Bugiel et al. "XManDroid: A New Android Evolution to Mitigate Privilege Escalation Attacks". In: *Technische Universität Darmstadt, Technical Report* (2011), pp. 1 –18.

[10] Rubin Xu et al. "Aurasium: Practical Policy Enforcement for Android Applications". In: *Proceedings of the 21st USENIX conference ...* (2012), p. 27.

[11] Giovanni Russello et al. "FireDroid: hardening security in almost-stock Android". In: *Proceedings of the 29th Annual Computer Security Applications Conference on - ACSAC '13* (2013), pp. 319–328.

[12] Michael Dietz et al. “Quire: lightweight provenance for smart phone operating systems”. In: *Proceedings of the 20th USENIX conference on Security* 271.2012 (2011), p. 23. arXiv: `1102.2445`.

[13] Xuetao Wei et al. “Permission Evolution in the Android Ecosystem”. In: *ACSAC '12 Proceedings of the 28th Annual Computer Security Applications Conference* April 2009 (2012), pp. 31–40.

[14] Giovanni Russello et al. “MOSES”. In: *Proceedings of the 17th ACM symposium on Access Control Models and Technologies - SACMAT '12*. New York, New York, USA: ACM Press, 2012, p. 3. ISBN: 9781450312950.

[15] Yifei Wang et al. “Compac: enforce component-level access control in android”. In: *Codaspy* (2014), pp. 25–36.

[16] Yajin Zhou et al. “Taming information-stealing smartphone applications (on android)”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 6740 LNCS.November 2009 (2011), pp. 93–107. ISSN: 03029743.

[17] Machigar Ongtang et al. “Semantically Rich Application Centric Security in Android”. In: *Acsac '09* August 2011 (2009), pp. 658–673. ISSN: 1063-9527.

[18] Yaser Mowafi et al. “A Context-aware Adaptive Security Framework for Mobile Applications”. In: *Proceedings of the 3rd International Conference on Context-Aware Systems and Applications - ICCASA '14*. ICST, 2014, pp. 147–153. ISBN: 978-1-63190-005-1.

[19] Xueqiang Wang et al. “DeepDroid: Dynamically Enforcing Enterprise Policy on Android Devices”. In: *Symposium on Network and Distributed System Security (NDSS)* February (2015), pp. 8–11.

[20] Michael Backes et al. “Boxify: Full-fledged App Sandboxing for Stock Android”. In: *24th USENIX Security Symposium (USENIX Security 15)* (2015), pp. 691–706.

[21] Earlence Fernandes et al. “FlowFence: Practical Data Protection for Emerging IoT Application Frameworks”. In: *Usenix Security* (2016), pp. 531–548.

[22] Android Open Source Project. *ART and Dalvik*. 2017. URL: `https://source.android.com/devices/tech/dalvik/index.html`.

[23] Xamarin Inc. *Android permission work-flow*. URL: `https://blog.xamarin.com/requesting-runtime-permissions-in-android-marshmallow/`.

[24] *How Xposed Works*. URL: `https://github.com/rovo89/XposedBridge/wiki/Development-tutorial{\#}how-xposed-works`.

[25] Jinyan Zang et al. “Who knows what about me? A survey of behind the scenes personal data sharing to third parties by mobile apps”. In: *Technology Science* 30 (2015).

[26] Android Open Source Project. *ActivityManager documentation*. URL: https://developer.android.com/reference/android/app/ActivityManager.html.

[27] Android Open Source Project. *Android Kernel Source*. URL: https://android.googlesource.com/kernel/msm/+/de9a6d36157b3eb10f72d6401a78277e3cc01a08.