TÉCNICO LISBOA

TrUbi

Mobile Operating System Security

Miguel Rodrigues Bento Barros da Costa

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. Dr. Paulo Jorge Pires Ferreira
Prof. Dr. Nuno Miguel Carvalho dos Santos

Examination Committee

Chairperson: Prof. Dr. José Carlos Alves Pereira Monteiro
Supervisor: Prof. Dr. Paulo Jorge Pires Ferreira
Member of the Committee: Prof. Dr. Henrique João Lopes Domingos

November 2015

Acknowledgments

I would like to thank my thesis supervisors, Professor Paulo Jorge Pires Ferreira and Professor Nuno Miguel Carvalho dos Santos, for the opportunity to work in such an interesting project such as this one, and for all the guidance provided throughout the whole process.

I would also like to thank my family, specially my parents, for all the support given, not only through this year but also throughout my entire life.

And finally my friends, which always managed to get me in a good and joyful spirit no matter what.

Abstract

Mobile devices, such as smartphones and tablets, are highly integrated in today's society. Due to their mobility, and wide variety of functionalities, these devices allow their users to perform a large set of diverse tasks of their daily life. Mobile devices allow the user to be constantly notified of events such as phone calls, texts, emails, while performing any other task as browsing the Internet. However, many usage scenarios require these functionalities to be constrained in some manner. As useful as it may be to be notified of said events, in a cinema, for example, it is in everyones' best interest that the mobile device be muted and such notifications silent. In this project, we present TrUbi, a system that allows for temporary restrictions of mobile devices by forcing some of their specific functions to be locked for a limited amount of time, e.g. sound muted, network access blocked, etc. To provide this restriction capability, TrUbi enforces global security policies by implementing an operating system primitive named trust lease. Our TrUbi prototype, implemented on Android OS, can efficiently prevent real-world apps from accessing devices' functions constrained by TrUbi and enable new mobile utilization scenarios that are currently unsupported, to the best of our knowledge, by existing operating systems. Lastly, the TrUbi prototype was also tested in terms of performance which registered a negligible impact on the overall system performance and energy consumption.

Keywords

Mobile Devices; Mobile Security; TrUbi; Trust Leases; Resource Restrictions; Android OS;

Resumo

Dispositivos móveis, tais como smartphones e tablets, encontram-se altamente inseridos na sociedade de hoje em dia. Devido à sua mobilidade, e grande variedade de funcionalidades, estes dispositivos permitem aos seus utilizadores realizar um conjunto bastante diversificado de tarefas do seu dia-a-dia. Estes dispositivos permitem aos seus utilizadores serem constantemente notificados de diversos eventos, entre eles, chamadas telefónicas, mensagens de texto, emails, tudo isto enquanto desempenham outras tarefas como procurar algo na Internet. Ainda assim, muitos cenários de uso requerem que estas funcionalidades sejam restringidas. Por muito vantajoso que seja ser notificado dos referidos eventos, no cinema, por exemplo, é do interesse de todos que os dispositivos se encontrem em silêncio, e que estas notificações sejam mudas. Neste projecto introduzimos o TrUbi, um sistema que permite restringir temporariamente as funcionalidades dos dispositivos móveis, obrigando que certas funções sejam bloqueadas durante um período limitado de tempo, por exemplo, desligar o som, o acesso à Internet, etc. De modo a proporcionar estas restrições, o TrUbi impõe políticas globais de segurança sobre o sistema através da implementação de uma nova primitiva do sistema operativo denominada trust lease. O protótipo do TrUbi, implementado em Android, é capaz de restringir de forma eficaz aplicações reais tendo por base as restrições requeridas. São assim proporcionados novos cenários de utilização não anteriormente suportados, segundo o nosso conhecimento, pelos sistemas operativos móveis actuais. Por fim o desempenho do protótipo foi testado mostrando que este regista um impacto negligenciável no desempenho geral do sistema operativo e no seu consumo energético.

Palavras Chave

Dispositivos Móveis; Segurança Móvel; TrUbi; Trust Leases; Restrição de Recursos; SO Android;

Contents

1 Introduction **1**

1.1 Motivation . . . 3

1.2 Contributions . . . 3

1.3 Other Solutions . . . 4

1.4 Results . . . 5

1.5 Structure of the Document . . . 6

2 Related Work **7**

2.1 Android Overview . . . 10

2.2 User-Centric Access Control Systems . . . 11

2.3 Mandatory Access Control Systems . . . 14

2.4 Application Interaction Control . . . 16

2.5 Privacy Enhancement Systems . . . 19

2.6 Access Control Hooks Frameworks . . . 22

3 Model and Architecture of TrUbi **27**

3.1 TrUbi - Trust Lease Model . . . 29

3.2 TrUbi Architecture . . . 30

3.3 OS Modes: Restricted and Unrestricted . . . 31

3.4 Trust Lease Restrictions . . . 32

3.5 Trust Lease Termination Events . . . 35

3.6 Remote Attestation of Trust Leases . . . 36

3.7 Summary . . . 38

4 Implementation **39**

4.1 TrUbi's Prototype on Android OS . . . 41

4.2 TrUbi's API . . . 44

4.3 Supported Trust Lease Restrictions . . . 45

4.4 Supported Trust Lease Terminators . . . 49

4.5 Trust Lease Persistence . . . 50

4.6 Summary 51

5 Evaluation **53**

5.1 Restrictions Enforcement Efficiency 55

5.2 System Overall Performance 55

5.3 Trust Lease Primitives Performance 57

5.4 Energy Consumption 60

5.5 Developer Friendly 61

5.6 Summary 63

6 Conclusion **65**

6.1 Conclusion 67

6.2 Future Work 68

List of Figures

2.1 Android Architecture. 10
2.2 CRePE's Architecture (Adapted from [1]). 13
2.3 Confused Deputy Attack Example (Adapted from [2]). 17
2.4 Saints architecture (Adapted from [3]). 19
2.5 Aquifer architecture for Android taken from [4]. 21
2.6 ASM Architecture from [5]. 23
2.7 ASF Architecture (Adapted from [6]). 24

3.1 Timeline of hypothetical mTheater strapp. 30
3.2 Trust lease technique 31
3.3 TrUbi high level architecture. 32
3.4 General representation of TrUbi's reference monitor. Light shaded boxes show strapp X that owns a time-limited trust lease for restricting network access. 33
3.5 Timeline of a typical trust lease: (1) strapp starts, (2) strapp invokes startlease, (3) OS enters restricted mode upon user authorization, (4) termination event triggered and callback executed, and (5) OS leaves restricted mode. 34
3.6 Time diagram of trust lease attestation protocol. 37

4.1 Architecture of TrUbi's prototype in Android OS: grey background indicates TrUbi components natively implemented; diagonal stripes under white background means that it was based on an unmodified ASM component; diagonal stripes under grey background denotes that it was modified from a pre-existing ASM component. 42
4.2 Lease Confirmer app. 43
4.3 Freezing routine of the process restriction. 46

5.1 Execution time of `startlease` with restrictions which require no OS action. 58
5.2 Execution time of `startlease` with restrictions which require simple actions from the OS. 59

5.3 Execution times of `startlease` with only one resource, camera. Scenarios with 1, 5 and 10 applications using this resource, respectively K1, K5 and K10. 59
5.4 Execution times of `quotelease` and `verifylease`. 60

List of Tables

2.1 Related Work systems' analysis. 25

5.1 Applications used to test the prototype restrictions enforcement efficiency. 56
5.2 Results of Softweg experiments. 57
5.3 Energy consumption overhead of TrUbi prototype. 60

Listings

4.1 Java code fragment to start a lease. 44
4.2 Example of a restriction policy written in XML. 45
4.3 Example of a lease's XML with a GPS terminator. 49
4.4 XML of a lease with a Wifi terminator 50
5.1 Java code of a simple ASM application to restrict the camera usage. 61
5.2 Java code of a simple strapp 62
5.3 XML file of a lease that restricts the camera (black) or applications (red) 63
5.4 Simplified code of an ASM application that restricts applications from running. 64

Acronyms

ADB Android Debug Bridge

AMS Activity Manager Service

AOSP Android Open Source Project

API Application Program Interface

API Application Programming Interface

ASF Android Security Framework

ASM Android Security Modules

ASM Android Security Modules

BYOD Bring Your Own Device

GPS Global Positioning System

IMEI International Mobile Station Equipment Identity

IMSI International Mobile Subscriber Identity

IPC Inter-Process Communication

IRM Inline Reference Monitors

LED Light-Emitting Diode

LSM Linux Security Modules

MAC Mandatory Access Control

OS Operating System

PMS Package Manager Service

SSL Secure Sockets Layer

TCB Trusted Computing Base

TCP Transmission Control Protocol

UI User Interface

UI User Interface

UID User Identifier

1

Introduction

Contents

1.1 Motivation 3
1.2 Contributions 3
1.3 Other Solutions 4
1.4 Results 5
1.5 Structure of the Document 6

1.1 Motivation

Increasingly, mobile devices like smartphones and tablets pervade our everyday lives. With a wide variety of functions and uses, these devices are a helpful tool in many different scenarios of the daily life. However, certain scenarios tend to be more restrictive than usual by requiring devices' functions to be constrained in some manner. For instance, in certain mobile scenarios, specific device features must be turned off. A simple example of such a scenario is the case of movie theaters. It is in the best interest of both movie attendees and movie theater owners that devices present inside the movie screening room are muted. Another useful restriction would be disabling the camera, preventing illegal recordings of said movies. Allowing personal devices to be used within enterprise contexts—so called Bring Your Own Device (BYOD)—also requires blocking access to specific resources. Many working environments require employees to have restricted Internet access. In certain corporate scenarios, mobile devices' microphone must be disabled. An example of these are privacy-sensitive meetings where there must not be any information-leakage. Likewise, restrictions may apply in other privacy-sensitive scenarios where disabling location sensors might also be necessary.

1.2 Contributions

The goal of this project is thus to develop, and implement, a novel security framework for commodity mobile devices, named TrUbi, that enables a restricted mode of operation. This mode of operation enables mobile devices to be turned into special-purposed devices, restricting their functionality to a well-defined set of tasks, for a limited amount of time. Besides the restrictions on the mobile device, the restricted mode also provides remote parties with guarantees that those restrictions are actually in place. TrUbi framework follows an application centric approach. This means that applications aware of its existence, named strapps, are able to request the activation of said restricted mode. The implementation of TrUbi's prototype was on top of the Android Operating System (OS) platform. The choice of Android was motivated by the fact that this operating system is the most used in terms of scientific and academic research on the field of mobile operating systems' security.

The success of the project also depends on the quality and practicality of the implementation. With this in mind, the final implementation must guarantee the following requirements:

Flexible Security Policies

Applications, using the restricted mode, must be able to properly specify all their security needs, i.e. security policies. TrUbi must therefore be able to provide flexible security policies. Mechanisms such as "Allow all but...", "Deny all but...", "Allow resource X", "Deny resource Y" need to be taken into consideration.

Internal and External Assurances

The TrUbi security framework must be able to assure, strapps and external entities, that the requested security policies are being enforced.

Efficiency

TrUbi must have a low impact on the overall system performance. The system must incur in low overheads while not operating under the restricted mode and a slight loss of performance is acceptable while the mobile device is in said mode.

Developer Friendly

Application developers must be able to specify, with little effort, and in detail, the security policies required by their applications, allowing them to focus on the logic of said applications rather than mobile specific security mechanisms.

This solution focuses on strapps' security needs. TrUbi puts the security needs of strapps before those of the system, and of other applications. In other words, strapps may request for an application, or even for the whole mobile operating system, to have its functionalities restricted. Strapps are able to request the enforcement of a set of security policies. This enforcement specializes the device in the execution of a very specific task, while granting behavior guarantees to internal and external parties. Said security policy enforcement is named a "Trust Lease" [7]. When faced with a trust lease request, TrUbi prompts the mobile device user for his approval on such a lease. The strapp is then notified of the user's response and can proceed accordingly.

In summary, the contributions of this project are:

- The design of TrUbi (Trusted Ubiquity), an OS independent security framework for enforcing application specific restriction policies on personal devices.
- Implementation, on an Android OS, of a prototype of TrUbi.
- Quality and viability assessment of the prototype.

1.3 Other Solutions

For some time now, mobile devices' security has been a subject of high interest in the scientific community. There has been many different studies and implementations of new security models that target these devices. Systems such as user-centric access control systems [1,8–10], mandatory access

control systems [11, 12], privacy enhancement systems [4, 13–15], access control hooks framework systems [5, 6] and application interaction control systems [2, 3, 16] are examples of some such systems. However, none of them, to the best of our knowledge, has ever directly associated the security needs of a security sensitive application and the remaining mobile devices' applications and resources. Applications are always seen as potentially misbehaving and harmful. The needs of the system, and of its users, always precede those of applications. This is the main reason why this project's goal is not achieved by any of the related work described in Chapter 2, in spite of their contribution in the development of this security framework.

1.4 Results

This project was successful in achieving its goals. TrUbi has now a concrete OS-independent architecture design (Figure 3.3), explained in detail in Chapter 3. This architecture is based on trust leases which allow for the restriction of the device based on strapps' security needs. Restrictions are applied to mobile devices' resources and settings. These restrictions are flexible and can limit the access of both applications and the operating system to said resources and settings.

Remote attestation is also guaranteed in TrUbi through the use of digital certificates. Essentially, TrUbi is the owner of a certified key pair which it uses to sign a digest of the currently being enforced trust lease. Since the OS is part of the Trusted Computing Base (TCB), i.e., it is assumed to have not been altered in anyway, this allows strapps, and remote entities, to be sure of the authenticity of the digest and therefore of the enforcement of the lease.

Developers are also taken into consideration as TrUbi exposes a set of four simple primitives. These primitives allow them to take full advantage of the system with little effort. The request of leases is done through the `startlease` primitive, which takes in the lease restrictions required by the strapp. The primitive `stoplease` can be leveraged by developers to terminate the lease ahead of its original termination event. Finally, `quotelease` and `verifylease` are the attestation primitives which allow developers to respectively request and verify the specification and authenticity of the lease being enforced.

Last, the implementation of TrUbi in its Android OS prototype allowed for a proper efficiency evaluation. Chapter 5 presents the benchmark results that prove that the TrUbi prototype has a small impact in the overall system performance. This is due to the type of enforcement and resources being enforced. TrUbi focus mainly on resources and settings which are only periodically used and usually through user interaction. This makes it so that the overhead of the extra lease permission verification is small and barely noticeable.

TrUbi and the corresponding prototype achieve this way all the requirements that this project proposed to fulfill, thus achieving its goal.

1.5 Structure of the Document

The remainder of this document proceeds as follows. Chapter 2 highlights the related work on mobile devices security. Chapter 3 describes TrUbi's model and architecture. Chapter 4 provides implementation details of TrUbi prototype in its current Android OS based version. Evaluation and viability aspects of the TrUbi prototype are presented and discussed in Chapter 5. Last, Chapter 6 highlights the most important aspects of the project.

2

Related Work

Contents

2.1 Android Overview 10
2.2 User-Centric Access Control Systems 11
2.3 Mandatory Access Control Systems 14
2.4 Application Interaction Control 16
2.5 Privacy Enhancement Systems 19
2.6 Access Control Hooks Frameworks 22

This chapter presents the related work, in particular, it focuses on systems aimed at improving mobile computing security. Although these systems could be coupled in several different categories, the chosen categories were so because they are aimed at very concrete security concerns of our system. Bellow is explained what is discussed in each category as well as which are the security concerns they address:

- **User-Centric Access Control Systems**
  Concern: How to restrict applications' access to mobile devices' resources at run time, and on demand.
  Overview: Systems which allow the user, at run time, to control which resources applications have access to.

- **Mandatory Access Control Systems**
  Concern: How to restrict resource access, in a fine grain manner, to the whole system instead of to a particular application.
  Overview: Systems which allow a fine grain control over mobile devices' resources, system wide, through the use of Mandatory Access Control (MAC) mechanisms.

- **Application Interaction Control**
  Concern: How to prevent untrusted and potentially misbehaving applications' use of secure applications' APIs.
  Overview: Systems that define mechanisms to restrict and control the interactions between applications.

- **Privacy Enhancement Systems**
  Concern: How to ensure the privacy of personal information stored in mobile devices.
  Overview: Systems which allow the monitoring and control of information leakage.

- **Access Control Hooks Frameworks**
  Concern: How to properly define flexible security frameworks that envision resource access control.
  Overview: Frameworks that place hooks in each and every mobile device's resource in order to have a fine grained control over them.

Besides the above stated categories, the Related Work section will also provide an overview of the

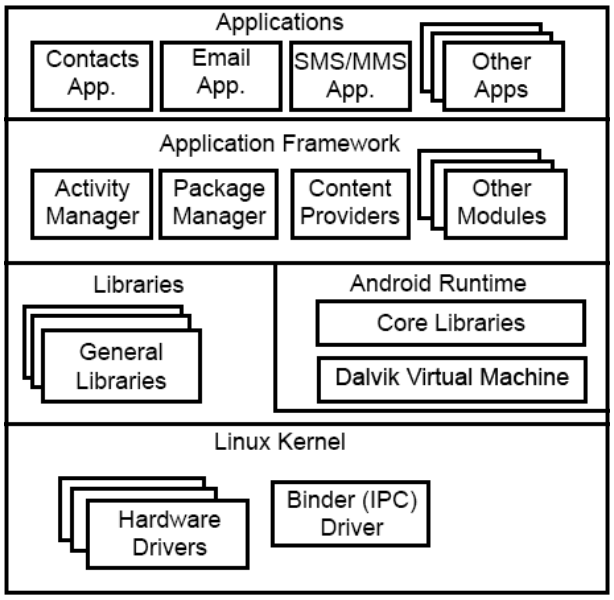


Figure 2.1: Android Architecture.

Android operating system. This overview allows for a better comprehension of the Android system, as well as how it keeps itself and its applications safe.

2.1 Android Overview

Android is a modern mobile operating system that provides an open source platform and application environment for mobile devices. Android's main building blocks [17] are (see Figure 2.1):

- **Device Hardware:** Android was designed to run on a wide range of hardware configurations. These configurations vary from smart phones and tablets to set-top-boxes.
- **Android Operating System:** The core operating system was built on top of the Linux kernel and all device resources are accessed through the operating system.
- **Android Application Runtime:** Most android applications are written in the Java programming language and run in the Dalvik virtual machine. However, core Android services and applications are native or include native libraries.

Android's application functionality is divided into four types of components: activities, services, broadcast receivers, and content providers. The activities compose the user interface of applications. Service components act as daemons, providing background processing. Broadcast receivers are responsible for handling asynchronous messages. Last, the content provider components are data servers, unique

to each application, that can be queried by other applications. Android allows its components to communicate with each other through the use of Binder Interprocess Communication (IPC).

To enforce security requirements Android uses permissions. Permissions are text strings that represent the mobile devices' resources and possible uses. Android applications are required to state, in a specific file called AndroidManifest.xml, which resources they require to run. These permissions are then granted to the application when it is first installed on the device and upon user confirmation. However, if the user denies to grant applications their full set of requested permissions, Android stops these applications from being installed into the user's device. Since every application has a unique Linux User Identifier (UID), Android can store each application's individual permissions in a compact way within a service called Package Manager Service (PMS). The enforcement of said permissions is mostly done by authorization hooks implemented within the Activity Manager Service (AMS). Each time an application tries to use a resource, a correspondent authorization hook is triggered which checks if the requesting application has the necessary permissions to use such resource. However, not all permissions are enforced using authorization hooks. Permissions that control access to low-level capabilities, such as opening network sockets or accessing the SDcard storage, are actually enforced at the kernel level.

2.2 User-Centric Access Control Systems

The systems addressed in this category focus on giving the user better control over their systems' resource usage by potentially ill-intentioned applications, when compared with the traditional, off the shelf, Android operating system. Among these systems we have MockDroid [10] and AppGuard [9] which allow the user to install applications with their requested permissions but support revoking or refining such permissions later on. Apex [8] and CRePE [1] systems, besides allowing run-time revocation and refinement, also enforce run-time environment based constraints. For Apex, constraints are, for example, the number of times a resource could be used, or the time of day when such resource usage is allowed. CRePE took these run-time constraints a little further and introduced the concept of context. Contexts, in CRePE sense, are defined by location, time of day, date, policy ownership, and other aspects that allow both a dynamic and descriptive analysis of the environment around the mobile device. However, in spite of all these systems having the same focus, their implementation of these security improvements vary from one another.

MockDroid [10] allows a user to, "mock", i.e. deny, a potentially malicious application's access to a resource. By mocking a resource, MockDroid provides a plausible but incorrect result to the application, thus achieving the desired goal. The resources that can be mocked are: location, internet, sms/mms, calendar, contacts, device ID and broadcast intents. This denial is done by deceiving applications, through the provisioning of bogus data, rather than explicitly denying them access to the resource in

question. This prevents mocked applications from terminating prematurely due to resource access exceptions. Since every resource behaves differently, to properly implement such deceptions, each and every resource must be addressed individually and in its own way. For example, the mocking of the internet resource may result in the application “thinking” that there is no available wireless network, where as the mocking of the device ID will result in a fake constant value being returned to the application. To control the access to the mobile device’s resources by potentially malicious applications, MockDroid modified Android’s native permission check so that it could intercept all resource access Application Programming Interface (API) calls from these applications. In truth, the native permission check was actually extended. When an application does a resource API call, besides its native behavior of checking if that resource was requested at install time, the system now also checks if the user has mocked the resource. In case the resource was not mocked, the API call completes in the same manner as the standard system. If it has indeed been mocked, fake data is provided back to the application and the goal of resource blocking achieved.

AppGuard [9] also focuses on restricting and revoking potential ill intentioned applications’ usage of the mobile devices’ resources. This system provides a set of built-in security and privacy policies aimed at the mobile system’s critical permissions. The internet policy, for example, allows a user not only to turn on/off the internet for a given application but also to specify which set of servers such application is allowed to connect to. The approach the developers of AppGuard took to implement these policies was based on Inline Reference Monitors (IRM). This technique allows the developers to rewrite the application binaries in order to invoke, at runtime, a security monitor before each security relevant program operation. This security monitor, dynamically checks whether the current policies allowed the attempted operation. If the operation is allowed, then the operation proceeds as in standard Android systems; otherwise, an alternative code is executed. This alternative code, for example, sometimes returns mock values to prevent the application’s termination due to exceptions.

Apex [8], as mentioned before, couples the revocation and refinement of security policies with dynamic variables such as time of day and other environment variables. This coupling allows users to have an even greater control over applications’ usage of mobile devices’ resources when compared with the previous systems. Apex developers’ main concern was to produce a system that allowed them to prove their concept, of security policies refinement with environment variables, to the scientific community. This lead the developers to implement a small system, with just a few constraints. Such constraints were the number of allowed uses of a resource and the time of day during which that resource could be used. For future work the authors had envisioned a desktop application which would allow expert users to write more environment constraints for their system. To associate dynamic events constraints and the security policies of the mobile device, Apex extended the native system’s permission framework and introduced the concept of ”Application Attributes”. Each application is given a set of attributes which

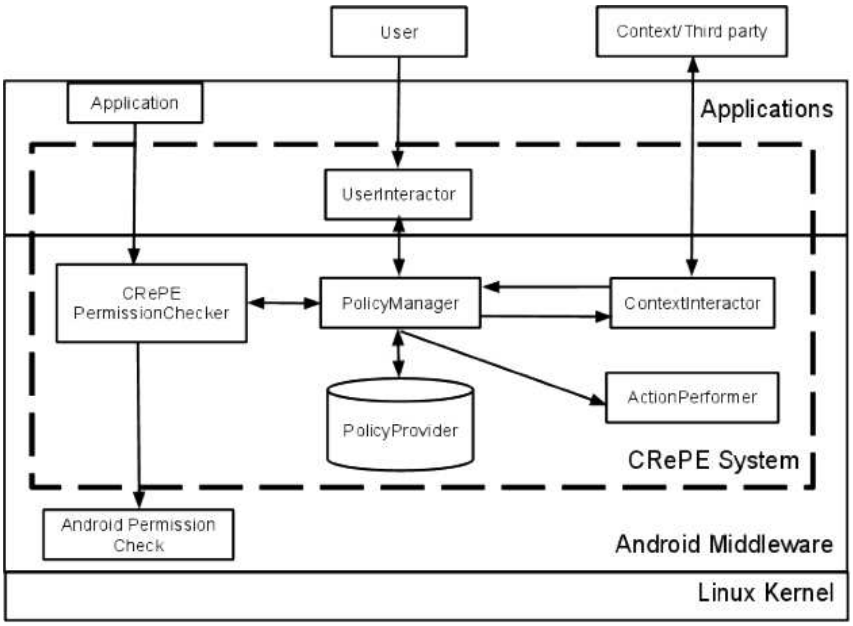


Figure 2.2: CRePE's Architecture (Adapted from [1]).

defines its state. A possible example of this state is the tuple: application ID, resource ID and max number of uses constraint, and corresponding value. This state is persistent and therefore kept during different system sessions. During the mobile device's usage a special attribute update function keeps the states of the applications up to date. These states are then used by a second permission function, defined by the developers of the system, in order to manage component calls with specific intents.

CRePE [1] provides a Context-Related Policy Enforcement for Android. This is by far the most ambitious system in this category of user-centric access control systems, when taking into account run-time environment constraints. This system incorporates the concept of context into itself. Contexts are abstractions of the environment in which mobile devices are on. They take into account factors like time of day, day of year and other such factors that influence the life of its user, and therefore can also influence his use of the mobile device. This allows users and third-parties to create date, time and location specific contexts that can be properly interpreted by their mobile devices. These contexts are then associated with a set of security policies that must be enforced each time their corresponding context is active. To implement the concept of context-aware constraints, CRePE divided its system into several modules, each one with a very small and precise set of purposes (See Figure 2.2). One of these modules is the "PermissionChecker". This module takes place before Android's native permission check. CRePE's permission checker intercepts requests sent from applications which try to start other applications, use system services, or its resources. Based on the active context and its policies, these requests are allowed or denied. The "PolicyManager" is the module that holds the list of the active contexts and their policies opposed to the "PolicyProvider" which stores all contexts and their corresponding policies

independently of them being active or inactive. Two other interesting components of CRePE are the "ContextInteractor" and the "ActionPerformer". These two modules are specially interesting due to the type of functionally they allow in the system. Each time a context becomes active, the "ContextInteractor" is the module that detects such change and informs the system. On the other hand, the "ActionPerfomer" handles the change of active policies at run-time. The cooperation between these two modules allows currently running applications to adapt to the change in contexts. Through their use, the system is able to enforce new restrictions, or disable old ones, upon already running applications. For example, lets say a company has a strict rule about the use of Facebook during its working hours. If a proper context and corresponding policy is set, upon entering the context of "working hours", which would be detected by the Context Interactor, the Action Performer module would then shut-down Facebook's application because a new set of policies was now being enforced. The last component of CRePE is the "UserInteractor". This module is an Android application which allow users to interact with CRePE and create, update and delete contexts and their policies.

Analysis: Because none of these systems' goals align with our own, none of them can be used to produce a system to achieve our goal. These systems consider the user, or third-parties, as trusted system managers, and the applications as potentially dangerous and miss-behaving. In contrast, our project considers the user to be a potentially malicious agent and requires the system to be managed according to the secure applications' needs. Since a user may alter applications permissions, at run time and without restrictions, it is impossible, with the above described systems, for a secure application to be assured that it will have all its required resources during its execution.

2.3 Mandatory Access Control Systems

As previously mentioned, many of the problems affecting today's mobile devices derive from the security polices present in their operating systems. More concretely, the problem usually resides in the coarse-grained security polices associated with applications' permissions to use system's resources. Having noticed such a fact, academic studies were made and systems developed addressing such problems. These systems [11, 12] tried to adapt fine-grain MAC techniques into mobile devices. The reasoning behind this was that those techniques were already successfully used in non-mobile machines when trying to enhance their base system's security. SEAndroid [11] and FlaskDroid [12] are two such systems that tried to adapt already successful MAC techniques and systems to mobile operating systems.

SEAndroid [11] brings flexible mandatory access control to Android operating systems. By providing flexible mandatory access control, SEAndroid is able to address several critical gaps in the security of today's Android running devices. To properly implement MAC mechanisms into Android systems,

SEAndroid developers set two main targets: First, and since Android is Linux-based, SEAndroid had to be able to enable an effective use of Security Enhanced Linux (SELinux [18]). SELinux is a mandatory access control mechanism for Linux. Second, SEAndroid needed to incorporate a set of middleware MAC extensions to the standard Android permissions model. In the end, both marks were met and SEAndroid is now able to fend off several known exploits of Android systems:

- **Root Exploits:** This kind of exploits escalate the privilege of an unprivileged application to gain full root access to the device. By achieving this exploit applications are then able to perform arbitrary actions on the device.

- **Application Vulnerabilities Exploit:** These exploits are actually vulnerabilities of legitimate applications that are used by ill-intentioned applications. These vulnerabilities usually allowed illegitimate applications to have access to sensitive data, or even modify security-relevant settings, without the user's authorization. Skype and Lookout Mobile Security were two such legitimate applications which had unprotected files that could be accessed by ill-intentioned applications.

FlaskDroid [12] also provides mandatory access control for Android-based platforms. To achieve this, FlaskDroid tackled two fronts. First, by defining an efficient policy language, inspired by SELinux, tailored to the specific needs of the mobile devices' middleware semantics. Second, by ensuring that the enforcement of such policies was done at both the middleware and kernel layers. Such enforcement was necessary because ill-intentioned applications usually explored both the middleware and kernel path, for resource usage, in order to try to avoid policy enforcement. FlaskDroid design was based on the Flask Architecture [19]. The Flask Architecture has a modular design that decouples policy enforcement from the security policy itself thus allowing multiple and dynamic, system wide, security policies to be supported. Through this design FlaskDroid achieved:

- **System-wide security framework**
  A security framework that operates both on the middleware and kernel layer. This addresses the previously stated vulnerabilities of policy enforcement bypassing.

- **Flexible security policies at the middleware layer**
  The mobile device's middeware layer incorporates now a new policy language. This new policy language was specifically designed for the rich semantics of this layer.

Analysis: Through the introduction of MAC techniques into mobile devices, these systems managed to fend off known exploits. However, their goal is always to protect the system and its applications from other ill-intentioned applications and third-parties. Still, these applications never distinguish between

secure and non-secure applications. They assume every application may be an ill-intentioned application and thus cannot have a negative impact in the execution of any other application. This fact alone renders these systems unusable to achieve our goal. In our project the most valuable objects are secure applications. It is mandatory that there is a distinction between secure and non-secure applications. Furthermore, all secure applications must have all their security needs fulfilled, independently of the cost paid by the system, or by non-secure applications.

2.4 Application Interaction Control

Mobile devices nowadays have a wide variety of applications and purposes. As more proficient users may use them for online sales and business trades, average users might stick with social media and other simple tasks. However, from one type of users to the other, mobile devices usually still carry a big set of applications, some even for the same purpose. Mobile devices' users have become used to having multiple choices and ways to fulfil their desired tasks. To allow such choices in a user friendly way, application developers have defined APIs in their applications giving other applications access to the services they provide. However, this flexibility and extra functionality introduced new security vulnerabilities. Ill-intentioned applications, with low system privileges, started to use trusted applications' APIs in order to fulfil actions and gather information that should not be reachable by them. This kind of attacks, called Confused Deputy Attacks, occur when an untrusted application (Requester) is able to trick a more privileged one (Deputy) into doing its own bidding. Figure 2.3 illustrates an example of a call chain where an untrusted application, which does not have permission to send SMSs, tricks a legitimate SMS application into sending a SMS on its behalf. A similar attack is the Collusion Attack. Collusion attacks focus on having two colluding applications with different resource permissions. Separately these applications are harmless, however, when colluding, i.e., working together, they can take advantage of each others permissions to compromise the mobile device's security. An example of this is an application that only allows the management of the device's contact list and nothing else, and an application with simple access to the internet but no permissions to access user data. Separately, they cannot divulge user information, however, when colluding, they can use each other's APIs to access the user's contact list and send it online without the user's knowledge.

These attacks, which are a form of privilege escalation exploits, are mostly allowed because the majority of mobile operating systems implement coarse-grain policies to regulate applications' APIs. Systems like Quire [16] and IPC Inspection [2] try to solve this problem by focusing on call chains, and by lowering the permissions of called applications to those of their callers. In the other hand, Saint's [3] approach was to extend Android's security architecture with security policies aimed at restricting applications' API usage.

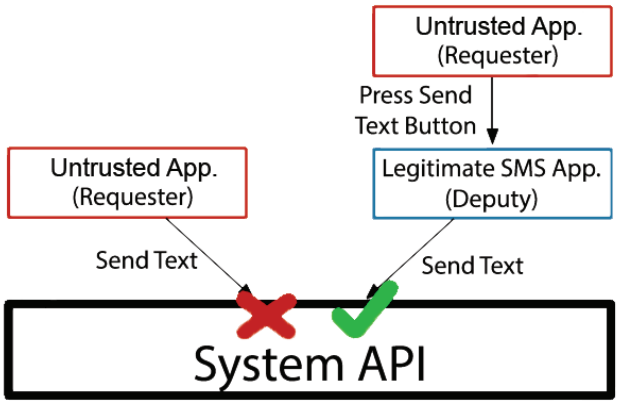


Figure 2.3: Confused Deputy Attack Example (Adapted from [2]).

Quire [16] approaches the problem of privilege escalation by engineering two new security mechanisms. First, and with the aim of providing authenticated call chains, a lightweight signature scheme allows applications to sign statements, conveying call chains' requesters and requests, that can be verified by any application on the same mobile device. Second, in order to prevent privilege escalation, Quire tracks call chains and forces applications to operate with the reduced privileges of its callers. Legitimate applications are still allowed to run on their full set of permissions, however, to do so they must act on their own. To implement these mechanisms, Quire extended Android's native Binder Inter-Process Communication (IPC) protocols. The extended version of these protocols automatically build calling contexts as call chains are formed. Through the use of calling contexts, Quire is now able to restrict resource access based on the caller with the lowest privileges. To do so, Quire has a module named IPC Provenance which has a set of endpoints, one by each sensitive resource, that checks call chains and reasons about the legitimacy of the request.

IPC Inspection [2] is another system which also focuses on lowering the permissions of called applications according to those of the caller. IPC Inspection is comprised of three primary mechanisms. First, it maintains a list of the current permissions of each applications, allowing called applications to know their callers both original and current permissions. Second, privilege reduction is built into the system's inter-application communication mechanisms making starting an application and sending a message both count as IPC. Last, receiving applications may accept or reject messages based on one of two methods: i) either the called application reasons its choice based on the intersection of both caller and callee's permissions, or ii) the called application sets an explicit list of allowed applications, rejecting all messages from those not in it. However, such mechanisms only work on clear communication relationships, meaning: the requester sends a message, or starts an application, and the deputy acts. Communications like request and reply, or duplex communications, can not be dealt the same way. This

means that IPC Inspection has to be able to distinguish between each type of communication. In the request-reply communications, IPC Inspection needs to reduce the privileges of the deputy to those of the requester; however, upon the reply of the message, the requester should not have his privileges reduced by the deputy. In the case of duplex communication, however, communication is a stream of data that flows between two applications (e.g., Transmission Control Protocol (TCP) sockets). With such a communication, applications can act both as deputy or requester so IPC Inspection must reduce the privileges of both.

Saint [3], which stands for Secure Application Interaction, although having the same problem in mind as Quire and IPC Inspection, implements a different solution. In order to avoid privilege escalation attacks like the confused deputy attack, Saint extends the current Android's security architecture with policies that address some key application requirements. Through extensive studies of mobile applications' development, three requirements were identified as being essential and still non-existent in today's mobile systems:

1. **Permission assignment policy.** Today's applications have limited ability to control to whom permissions for accessing their interfaces are granted. It is mandatory that applications, which offer APIs for their services, can control which other applications are able to use them.

2. **Interface exposure policy.** Mobile devices provide rudimentary facilities for applications to control how their interfaces are used by other applications. By allowing applications to control how their APIs can be used, trusted applications can avoid being used by untrusted ones for illegitimate reasons.

3. **Interface use policy.** Applications have limited means of selecting, at run-time, which application's interfaces they use. Unlike the confused deputy attack where the caller is untrusted, the other way around may also occur. Callers may be forced to use an untrusted API due to the necessity of use of the service they provide. In order not to compromise the callers functions, applications must also be allowed to select which application's APIs they use.

Saint's architecture, illustrated in Figure 2.4, has four main components, Saint Installer, Saint Mediator, AppPolicy Provider, and FrameworkPolicyManager. The Saint Installer module is a modification of the native operating system application installer. This installer receives the path for the application and parses its configurations. Aspects like application signature, requested permissions and application version are gathered by this parsing. Another module, Saint Mediator, is a runtime enforcement module which covers four critical component interactions: starting new Activities, binding components to Services, receiving broadcast Intents, and accessing Content Providers. In order to mediate such interac-

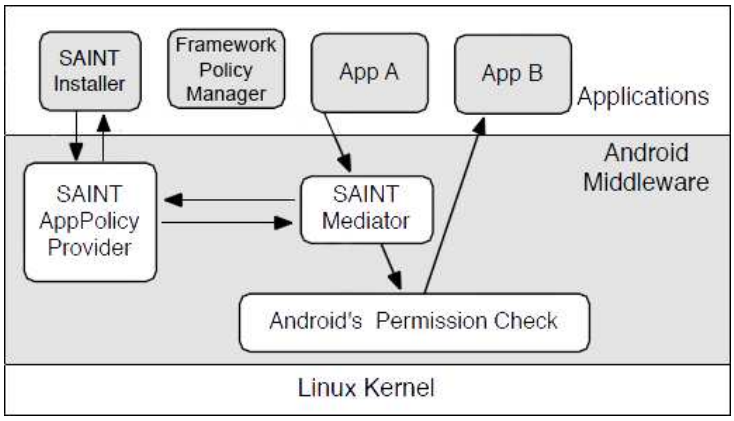


Figure 2.4: Saints architecture (Adapted from [3]).

tions, security modifications and authorization hooks had to be implemented into the native system. The third module being described, AppPolicy Provider, is responsible for storing both install and run-time policies. More importantly, this module is the policy decision point. At install-time, the Saint Installer passes the information about the application being installed to the AppPolicy Provider for the decision making of whether or not to install such application. The decision is based on whether there are conflicts with the active policies stored in this module and those of the application. Finally, the FrameworkPolicyManager module allows the user to disable, through the use of an interface, policies previously set by applications during their installation. This module is implemented as a normal application and is responsible for updating policies that the user has disabled/enabled. To do so, the module uses an AppPolicy Provider API, called updateApplicationPolicy, which can only be used by the FrameworkPolicyManager module.

Analysis: Just as in the previous sections, the above described systems are also not fit to accomplish our goal. With their sole focus on application interaction and privilege escalation prevention, these systems fail to grasp the need to define secure and insecure applications. Once again the user is the main pillar of the system's security and all applications are seen as potentially ill-intentioned. The worst possible scenario, which is common in all presented systems of this section, is when an application obtains more resources than those originally given by the device user. At no point in their project did these systems' developers consider the mobile devices' users as potentially ill-intentioned users that aimed to break secure applications.

2.5 Privacy Enhancement Systems

Nowadays mobile devices are used to save and keep track of many aspects of its user's life. Due to its mobility and ease of use, mobile devices are the usual go to when it comes to storing personal data. However, one of the challenges of today's mobile devices is keeping such sensitive data safe.

The systems presented in this section allow us to understand how today's security measures, within off the shelf commodity mobile devices, are inefficient and allow sensitive and private information to be leaked. The reason of such inefficiency is essentially due to the fact that most devices implement coarse-grained security policies. This type of security policies grant access to system resources and from there on applications may use them as they see fit. These policies make it possible for applications to ask for a resource in a legitimate way and then use it for ill-intentioned purposes. For example, a news feed application, which requires access to a mobile device's state and internet connection, may do so without raising any user suspicion. By stating that the internet resource is required for the gathering of news, and that the state is for notification purposes, the user would be compelled to accept such terms. However, many such applications do indeed gather and share private information covered by the veil of legitimacy. With the two previously stated resources, one application could, and many do so, share the users' phone number, International Mobile Subscriber Identity (IMSI) and International Mobile Station Equipment Identity (IMEI). In order to prevent misuse of mobile devices' resources, several systems focus on detecting how coarse grained policies are allowing sensitive information to leak and how can this leakage be stopped or at least be noticed. The above described systems could be divided into two separate groups: Information Leakage Tracking and Information Leakage Blocking. TaintDroid [13] belongs to the first while AppFence [14], TISSA [15] and Aquifer [4] belong to the latter.

TaintDroid [13] focus on tracking private information flows throughout the mobile device operating system. This system is particularly useful to understand how miss-behaving applications gather and share users' private data. In order to properly track information flows, TaintDroid identifies and tracks several sensitive information on the mobile operating system. The particular method used is a technique called "tainting". Tainting essentially refers to the act of marking an object. This technique attaches a marker on all sensitive data, present in the mobile device, and propagates this marker everywhere this data is sent to. An example of sensitive data could be the users' contact list, written messages or even its geographic location. The taint is set at the message level; this prevents devious receivers from unpacking variables in different ways in order to avoid the propagation of the taint.

AppFence's [14] main concern is to prevent data leakage. To achieve this, AppFence uses an extended version of TaintDroid that allows AppFence to detect unauthorized information flows. Unwanted data propagation is stopped through the use of two possible methods: either by shadowing or by exfiltration blocking. The first, shadowing, is a technique that replaces the original data with fake data or with none at all. The second, exfiltration blocking, is achieved by blocking the communication channel by which the propagation is going to take place and by stating to the application that such resource is not available. Both methods have faults and may break the original application; however, when one method breaks the application, the other is usually enough to stop the information while not breaking it. This fact made it so that AppFence uses both methods in a mixed methodology thus accomplishing better results.

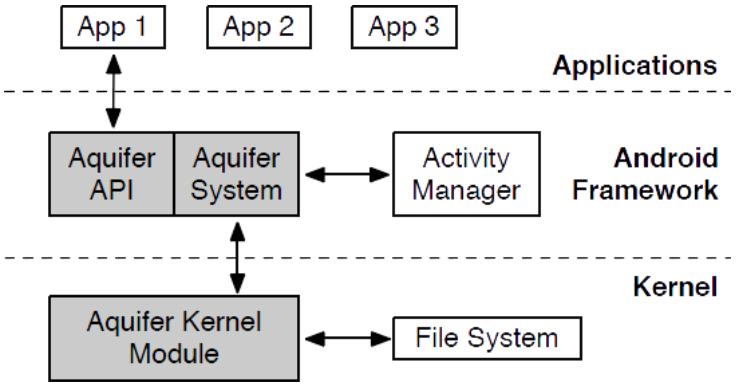


Figure 2.5: Aquifer architecture for Android taken from [4].

TISSA [15] also focuses on enhancing the privacy controls of mobile operating systems. However, this system does it so by defining new policy models rather than by information tracking and control like the previous one. In TISSA, the user is allowed to chose, at run-time if necessary, and in a fine-grained manner, which personal information applications may have access to. Unlike coarse-grained methods, that give full access to a given resource, fine-grained ones require very specific and detailed enumeration of what can be accessed. For example, resources such as device ID, contacts, call logs and location must be explicitly allowed by the user in order for an application to be able to use. Once again the implementation of such techniques require the extension of the native permission model of the mobile device. Through the design of mediators, named privacy-aware components, TISSA controls all applications' requests to sensitive data and allows or denies them based on their previously set fine-grained permissions.

Aquifer [4] also uses a different approach than that of the previously stated systems. To limit the sharing of personal information, this system uses user interface workflows. User interface workflows are an abstraction of several User Interface (UI) screens, that may or may not belong to the same application, but which share the same user intended task. The required security policies are then applied to such workflows rather than to a single application, allowing the system to control the sharing of personal information through several applications and limiting its leakage. To identify workflows, Aquifer requires the users to "use" their mobile devices. Workflows are defined, at run-time, as the users actually use the applications present in their devices. Several applications may be crossed while the user is executing a single task. User actions like sharing a photo may require the user to access the media gallery application followed by a media sharing application that will upload the photo to the internet. To implement access control Aquifer adds three modules to the native Android system, as illustrated in Figure 2.5. The first, the Aquifer API, which is used by the applications instead of the Android's usual resource API. The second, the Aquifer system, per say, that communicates with the API and the Android's activity manager. This module relates the active security policies with the current workflow and

the resources requested by the running applications. Last, the Aquifer Kernel Module, which mediates accesses to the mobile device's file system.

Analysis: The previously described systems, although efficient in blocking information leakage, are still unable to achieve our goal. Once again the user is viewed as a trustworthy entity and therefore never called into question. Consider a malicious user that works in a corporate company. Consider that this user's normal behaviour is to attend meetings and record them with his phone in order to write the minute upon its end. This section's systems would indeed be able to protect the recording from other applications. However, they would be unable to protect it from the phone's user. The user could just send the recordings to whomever he pleased. With TrUbi however, if the company developed a simple secure recording application with a proper security policy, such illicit propagation of a company meeting would not be allowed.

2.6 Access Control Hooks Frameworks

These systems, or better yet, frameworks, try to bring together the common needs of several researchers, and system developers, who have been studying and enhancing the current mobile operating systems' security models. Most of the systems that aim to achieve better control over the resources used by applications, and still allow flexibility, realized that: it is necessary to place hooks on the systems' resources APIs in order to actually control when these are going to be used. This, the main drive for the frameworks presented in this section is to provide programmable interfaces that allow resource access control based on authorization hooks. These interfaces are then to be used to develop security a rchitectures potentially leading to enhancements in the field of mobile devices' security. Android Security Modules (ASM) [5] and Android Security Framework (ASF) [6] are two such frameworks. Although both are conceptually very similar, they differ in the approach taken. Whereas ASM seeks to ensure the existing sandboxing guarantees of Android, ASF allows third-parties to extend Android's security framework potentially breaking those guarantees. This means that ASM can only make enforcement more restrictive, i.e. fewer application permissions or less file system access. Next we explain how these frameworks were implemented and how the difference between them shaped their architecture.

ASM [5] developers, besides the goal of providing a programmable interface, also required that their solution did not need root access to work properly. This requirement was envisioned so that the final product could be adopted by Google further enabling security enhancements in the field of mobile computing. The ASM framework provides a foundation for building reference monitors which are essential to ensure secure operating systems.

In order to better guide the design of ASM, the developers state a set of goals that must be followed:

G1: Generic authorization expressibility: Allow both prior and future security enhancements for mo-

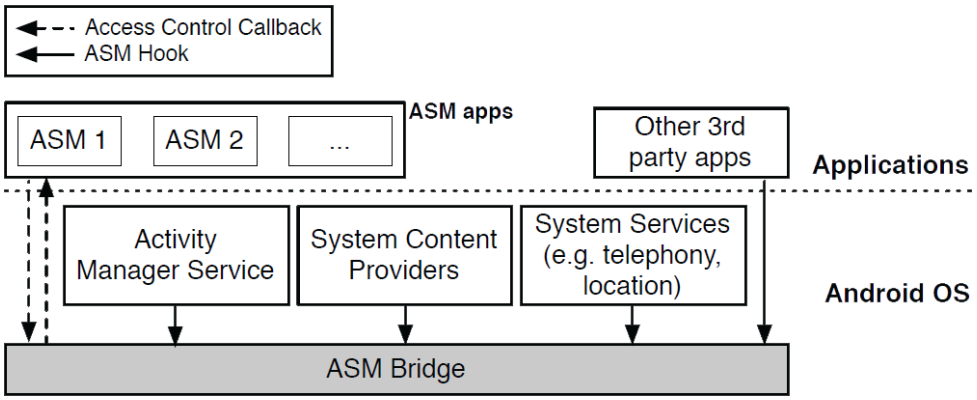


Figure 2.6: ASM Architecture from [5].

bile devices.

G2: Ensure existing security guarantees: Reference monitors should only make enforcement more restrictive.

G3: Protect kernel integrity: The kernel hooks interface cannot be provided to third-parties without some kind of control.

G4: Multiple authorization modules: Careful design of the framework's architecture to address potential conflicts in coexisting security modules.

G5: Minimize resource overhead: When no authorization module is loaded, the framework system should have negligible impact on the mobile device's system resources.

ASM also assumes that the base operating system and services are trusted. This assumption defines ASM's TCB as including the kernel, the AMS system itself, the Package Manager Service (PMS) and all OS services and content provider components. Figure 2.6 shows the architecture of ASM. Reference monitors are implemented as ASM applications. Each one of these ASM application registers for a unique set of authorization hooks, specifying a callback for every hook present in that set. When protected operations occur, ASM automatically invokes the call back routines defined in those ASM applications. The reference monitor interface provided by ASM is supported within the ASM Bridge. The ASM Bridge is responsible not only for managing the ASM apps, but also for receiving protection events from authorization hooks placed throughout the mobile operating system. These authorization hooks are then forwarded to the ASM apps that registered for the hook. An important aspect though, is that hooks only notify the ASM Bridge if they have been explicitly enabled. This allows the system to achieve one of its goals, G5, minimized resource overhead.

ASF [6], similarly to ASM, also provides a security API that supports authors of security extensions in developing their modules. However, unlike ASM, ASF allows developers full control over the operating

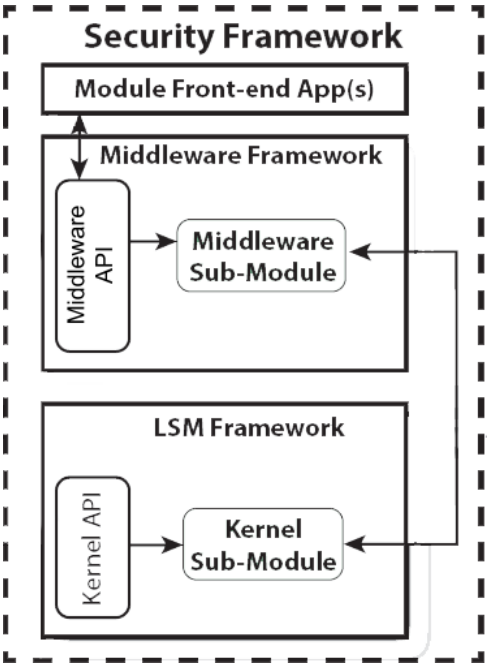


Figure 2.7: ASF Architecture (Adapted from [6]).

system's resources. This aspect of ASF, although adding expressibility, can potentially break the sandboxing guarantees of the Android System. To better achieve the desired goal, ASF's development also followed a strict set of design principles:

1. **Provisioning of policies as code instead of data:** Policies should be supported as code instead of data, just like rules written in one predetermined policy language. Providing an extensible security framework that supports loading of policy logic as code avoids committing to one particular security model or architecture.

2. **Providing a policy-agnostic OS security infrastructure:** The security framework and its API should be policy-agnostic. Meaning that, different layers of the software stack are aware of the security infrastructure. However, policy-specific intrusions into these layers are avoided and policy-specific data structures and logic are confined to security modules.

In Figure 2.7 we can see the overall structure of ASM. This framework is divided in three main modules: Module Front-end App(s), Middleware Framework and Linux Security Modules (LSM) Framework. The LSM Framework implements an infrastructure for mandatory access control at kernel level. In addition, it provides a number of enforcement hooks within kernel components, such as process management, network stack, and virtual file system. Operational interdependencies, between kernel and user spaces, sometimes require the implementation of proprietary channels of communication between these spaces. One such case could require kernel access control decisions to be propagated into the Middleware

Systems\Requirements	Flexible Security Policies	Internal Assurances (Applications)	External Assurances (Third Parties)	Efficiency	Developer Friendly
MockDroid [10]	✗	✗	✗	✓	✗
AppGuard [9]	✓	✗	✗	✓	✗
Apex [8]	✓	✗	✗	✓	✗
CRePE [1]	✓	✗	✓	✓	✗
SEAndroid [11]	✓	✗	✗	✓	✗
FlaskDroid [12]	✓	✗	✗	✓	✗
Quire [16]	✗	✗	✗	✓	✗
IPC Inspection [2]	✗	✗	✗	✓	✗
Saint [3]	✗	✗	✗	✓	✓
TaintDroid [13]	✗	✗	✗	✓	✗
AppFence [14]	✓	✗	✗	✓	✗
TISSA [15]	✓	✗	✗	✓	✗
Aquifer [4]	✓	✗	✗	✓	✗
ASM [5]	✓	✗	✓	✓	✓
ASF [6]	✓	✗	✗	✓	✓
TrUbi	✓	✓	✓	✓	✓

Table 2.1: Related Work systems' analysis.

Sub-Module. The Middleware Framework extends system services and applications that implement the operating system API. This extension applies hooks that enforce access control decisions made by the Middleware Sub-Module. The Middleware Sub-Module is a management application that includes the new security API. This module is deployed in a protected location on the file system, from where it is loaded during boot. The Front-end Apps refer to standard Android applications that act as front-end applications and communicate through the framework's middleware API directly with the middleware module.

Analysis: It is important to notice that, as frameworks, these systems can not fulfil the desired goal. Both ASM and ASF provide security mechanisms that although useful for the development of a project such as TrUbi, they are but a component of the set required to achieve it. Both frameworks allow the development of "special applications" which can enforce security policies. However, there is no atomic link between these policy enforcement applications and the typical Android applications. As they are, ASM and ASF would require developers to produce two separate applications: The policy enforcement application, meaning that developers would have to know how to write security policy enforcement routines, and the actually desired Android application. TrUbi strives to be developer friendly, allowing developers to focus on their applications and be assured that the system will provide them with the security they need.

Summary

After the analysis of all these systems and frameworks throughout the section of Related Work it is possible to conclude that the goal of this project is quite unique. While most scientific and academic research focus on protecting mobile devices, and their users, from ill-intentioned applications, our is to protect secure applications from ill-intentioned users and insecure applications. This focus proved to be a new paradigm in mobile security and so made impossible to find, to the best of our abilities, systems that shared the same goal. However, it is our belief that through this new paradigm for mobile security, new and improved interactions will take place between mobile devices and their users, further increasing the usefulness of mobile devices.

3

Model and Architecture of TrUbi

Contents

3.1 TrUbi - Trust Lease Model 29
3.2 TrUbi Architecture 30
3.3 OS Modes: Restricted and Unrestricted 31
3.4 Trust Lease Restrictions 32
3.5 Trust Lease Termination Events 35
3.6 Remote Attestation of Trust Leases 36
3.7 Summary 38

This chapter focuses on TrUbi's (Trusted Ubiquity) architecture, an OS independent security framework for enforcing application specific restriction policies on personal devices. As mentioned before, the goal of this project is to develop, and implement, a novel security framework for commodity mobile devices that enables a mode of operation called restricted mode. While in this mode, and for a limited amount of time, devices have their functionalities restricted to a well-defined set of tasks. Furthermore, during this period, interested remote entities may request this security framework for proper attestation of said restrictions. This type of attestation is possible because the OS is assumed to not have been tampered with, making it part of our TCB.

The above stated requirements are delivered by TrUbi and explained in proper detail in the following sections. First, Section 3.1 explains the whole TrUbi concept, including the trust lease [7] model. Next, Section 3.2 illustrates the global architecture of TrUbi. Section 3.3 describes how TrUbi introduces this restricted mode. Trust lease restriction rules are then explained in Section 3.4, focusing on their different types and consequences. Section 3.5 will entail the termination events that dictate the end of the lease's duration. Lastly, Section 3.6 explains the process of remote attestation, allowing this security framework to provide proper attestation to remote entities.

3.1 TrUbi - Trust Lease Model

As mentioned before, TrUbi is a trust lease security framework, OS-independent and with the goal of enforcing application specific restriction policies on personal devices. In order to enforce such restrictions without depending on trusted device administrators, and while being able to adapt to ever changing usage scenarios, trust leases take an application-centric approach. This means that this model relies on strapps to decide which restriction policy to apply. Strapps are essentially applications that are aware of this security framework and are therefore able to leverage it into allowing them to request the enforcement of global restriction policies on the device.

For a better understanding of these concepts consider the example of a cinema. Cinema attendees are constantly being bothered by other attendees' mobile devices sound. To reduce the occurrence of this type of annoyances one can build a strapp named mTheater. This strapp, upon admission to the screening room, sets a restriction policy that sets the device in silent mode until the movie ends. Figure 3.1 depicts a simplified timeline of mTheater's workflow. First the user starts by installing mTheater (Step 1). From then on he can buy cinema tickets which are stored in his device in their digital form (Step 2). At the ingress of the screening room, the user validates his ticket by communicating with an NFC terminal (step 3). In addition to validating the ticket, mTheater requests the OS to issue a trust lease for the intended restriction policy. The OS pops up a dialog message (step 4) asking the user whether he accepts the conditions of the trust lease, i.e., to mute the device, for the duration of the

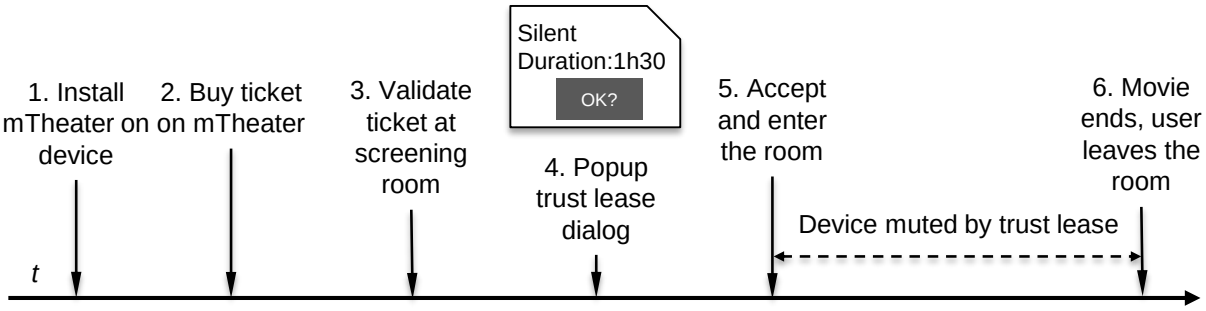


Figure 3.1: Timeline of hypothetical mTheater strapp.

movie. Assuming that the user accepts (step 5), the trust lease is issued, and the device restricted. When the movie ends, the lease automatically expires (step 6). If for any reason, upon presented with the dialog message of step 4, the user does not accept the terms of the lease, actions are taken to deny him access to the screening room. Such actions can be either not allowing him to physically enter the room, or even invalidating his cinema ticket.

The trust lease abstraction, illustrated in Figure 3.2, captures the process of restricting the functionality of a mobile device. It can be seen as a contract between the device's user (the lessor) and an interested party (the lessee). Lessees can be either a specific security sensitive application, or a remote entity which requires specific security guaranties. This contract defines a restriction policy that specifies the functions of the device that must be disabled and the condition events that cease these restrictions (typically a time limit). While a trust lease is active, the user cannot override its restrictions. To ensure that the user keeps his device under control, he must always approve trust leases before they are issued. Furthermore, the trust leases' restrictions are not permanent. The OS defines a maximum trust lease duration that must be verified before issuing the lease. In mTheater, for example, the movie theater company represents the lessee, and the restriction policy is represented in Figure 3.1.

3.2 TrUbi Architecture

Figure 3.3 illustrates a high level architecture of TrUbi. The so called normal applications, i.e., the applications that used to run on the original operating system, do not need to be altered in any way to work on top of TrUbi. For all matters and purposes, TrUbi is completely transparent to these applications. Strapps on the other hand, although acting as normal applications, know of the existence of TrUbi and can therefore request the enforcement of global restriction policies on the device. Among other actions, these global restrictions policies can limit the behavior of running applications, mobile devices' resources (e.g. network and camera) and their settings (e.g. sound and screen brightness). These restrictions are achieved by issuing trust leases on the running OS. Strapps request trust leases through TrUbi's

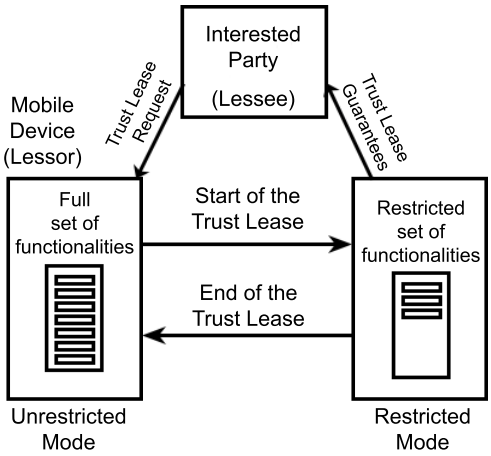


Figure 3.2: Trust lease technique

API, more concretely, using the `startlease` method further explained in Section 3.3. TrUbi then uses its module named Lease Confirmer to present a dialog message asking the user's agreement on the lease's conditions and enforcement. If the reply is negative, the process is canceled. Otherwise, TrUbi saves the lease in its Lease Store module and begins its enforcement. From here on, applications, resources and settings have their behavior restricted by TrUbi that does so based on the currently active leases. Essentially, before an application is started, a resource accessed, or a setting changed, each of these modules uses the TrUbi Bus to communicate with TrUbi and ask if such an action is allowed.

To achieve this type of enforcement TrUbi leverages a reference monitor architecture (Figure 3.4). The reference monitor maintains the state information about trust leases and coordinates their enforcement in the system by interacting with restrictor and terminator components. Restrictors implement specific restriction rules by setting the initial state of targeted system modules, i.e., resources and settings, and preventing unauthorized modifications to that state until the trust lease ends. Terminators are responsible for implementing policies' termination rules (as explained in Section 3.5). Each trust lease maintains a reference to the strapp that requested its creation, commonly called as the lessee strapp. As explained in sections 3.5 and 3.6, only the lessee strapp can invoke certain trust lease operations, namely `stoplease` and `quotelease`.

3.3 OS Modes: Restricted and Unrestricted

Strapps may instantiate trust leases. When a trust lease is instantiated, the OS applies the necessary restriction policies p, causing the device to enter a restricted mode state, i.e., a state where some functions (e.g. volume and brightness control) are temporarily disabled (Figure 3.2). When the trust lease expires, before switching back states, the system invokes the callback function f, which enables the strapp to perform any cleaning operations (e.g., encrypt data) deemed necessary. After these cleaning

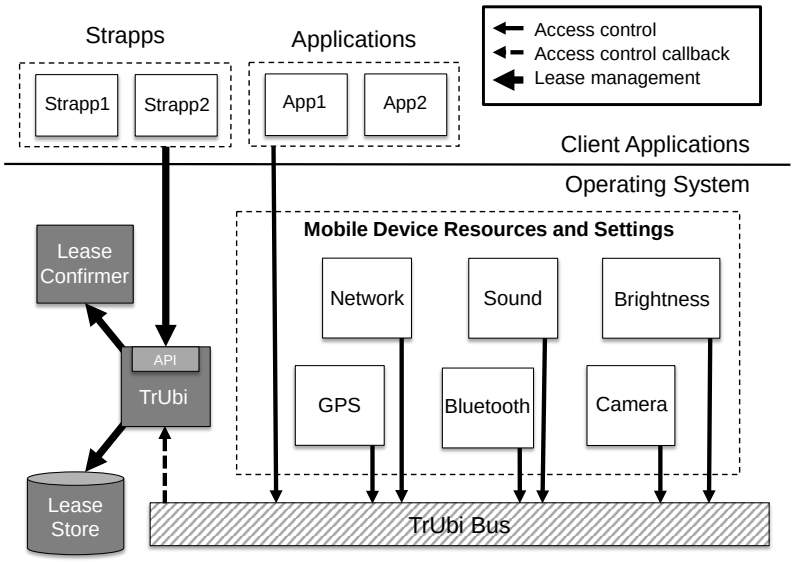


Figure 3.3: TrUbi high level architecture.

operations the restrictions are removed, and the device switches back to unrestricted mode.

To instantiate a trust lease and enter restricted mode, a strapp developer invokes:

$$\text{startlease}(p,\ f) \rightarrow l_{id} \mid \text{FAIL}$$

This system call takes a restriction policy p and a callback function f. The restriction policy comprises: restriction rules (Section 3.4) for defining the functions to be disabled, and termination rules (Section 3.5) for specifying the termination conditions of the trust lease. This procedure of switching from unrestricted to restricted, and back again, follows a typical timeline illustrated in Figure 3.5.

3.4 Trust Lease Restrictions

Restriction rules define states of one or more system resources or settings that cannot occur while a trust lease is active. For example, an hypothetical restriction named “Silent” to control the sound volume will mute the sound when the trust lease starts, and keep that configuration unchanged until the trust lease expires. Restrictions are application driven and for this matter TrUbi supports both white and blacklists. This means that if a whitelist is used, then all applications are denied except those on said list. On the other hand, if a blacklist is used, all applications are allowed except those on the blacklist. Depending on the kind of module the restriction rule refers to, TrUbi defines three major types of restrictions. We now explain these three major types and the envisioned restrictions for each of them.

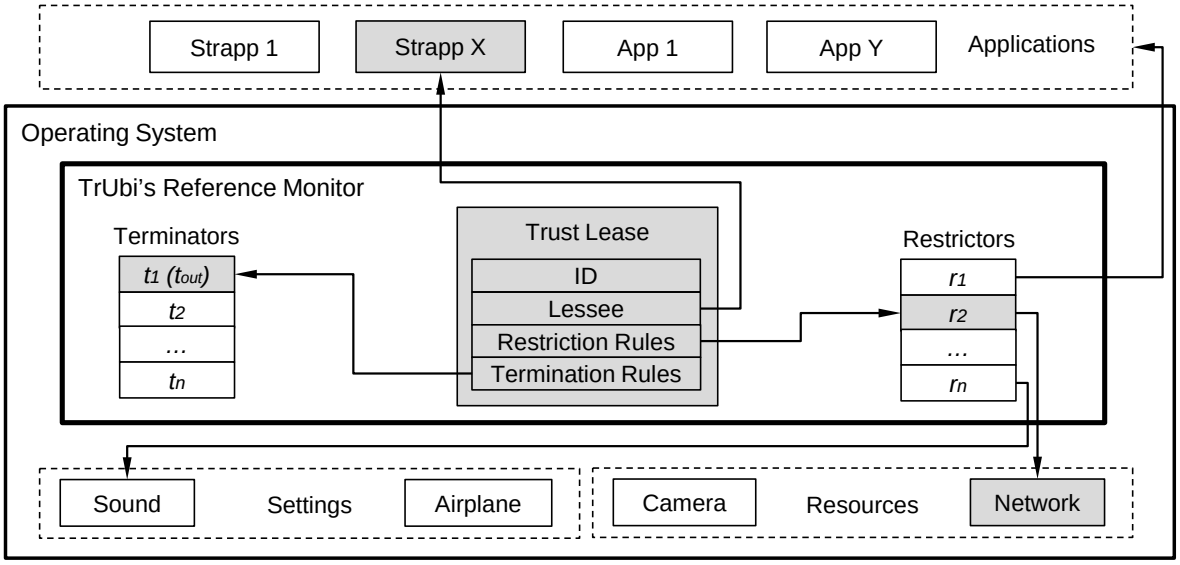


Figure 3.4: General representation of TrUbi's reference monitor. Light shaded boxes show strapp X that owns a time-limited trust lease for restricting network access.

Applications Restrictions: Applications restrictions enable blocking the execution of specific applications based on the name, version, developer identity, or other application identity elements (e.g. App Digest).

- **Process:** Prevents the execution of applications (e.g. facebook, instagram, snapchat, etc). When a trust lease starts, TrUbi checks all running application processes for proper authorization and suspends all those that do not have one. While the lease is active, TrUbi forbids the execution of new processes that do not have been granted execution privileges. When the lease terminates, TrUbi resumes all previously suspended processes. By suspending and resuming applications' processes, TrUbi aims to cause minimal impact to the user's experience.

Resources Restrictions: Resource restrictions control access to peripherals (e.g., network, or camera) or services (e.g., screenshot service, etc.). When used, global access to specific resources can be disallowed.

- **Network:** Prevents access to the Internet. When a lease is enforced with this restriction, TrUbi checks all ongoing Internet connections and terminates all which have not been granted proper authorization. During the lease enforcement, unauthorized applications are denied the ability to start new connections as well.
- **Camera:** Denies access to the Camera. Applications accessing the camera have their access

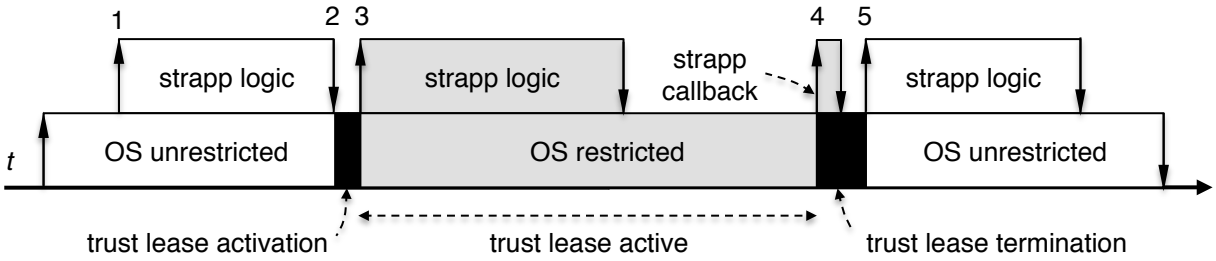


Figure 3.5: Timeline of a typical trust lease: (1) strapp starts, (2) strapp invokes startlease, (3) OS enters restricted mode upon user authorization, (4) termination event triggered and callback executed, and (5) OS leaves restricted mode.

revoked and future accesses are denied to all those that have not being granted access by the lease specification.

- **Bluetooth:** Prevents access to the Bluetooth resource. When a lease is enforced with this restriction, TrUbi checks all ongoing Bluetooth connections and terminates all which have not been granted proper authorization. During the lease enforcement, unauthorized applications are denied the ability to start new connections as well.
- **Phone Call:** Prevents applications from making or receiving phone calls. During the enforcement of a trust lease new calls are denied and ongoing calls terminated.
- **SMS & MMS:** Prevents sending and receiving SMS and MMS messages.
- **Screenshot:** Blocks the ability of a mobile device of taking screenshots of the contents present in its screen display. This restriction is specially useful for sensitive content sent through applications such as Snapchat.
- **Microphone:** Denies access to the microphone of the mobile device if the application has not been granted access to it by the trust lease.
- **LED:** Most mobile devices have a built in camera with a Light-Emitting Diode (LED) for flash purposes. This rule forces the camera built-in LED to be activated only by applications which have been granted access in the trust lease specification.

Settings Restrictions: Settings restrictions refer to system configurations, such as muting the device, enabling flight mode, etc. These configurations are typically exposed to the user through specific system services.

- **Sound:** Sets an initial value for the device's volume and forces it to be kept unchanged, unless the application has the proper access rights.

- **Brightness:** Allows the trust lease to set an initial value for the brightness of the device's screen and prevents changes to this configuration by the user or by applications which have not been given proper authorization.
- **Airplane Mode:** Allows the lease to set an initial value for this setting, namely on or off, which is kept unchanged throughout the duration of the lease.
- **Time:** Prevents changes to the system's date, time and timezone.

3.5 Trust Lease Termination Events

When a trust lease is active, the OS remains in restricted mode until a termination event occurs. Termination events are delivered in one of three ways:

Time Based: Typical termination rules define time conditions, e.g., a timeout (T_{out}). When a trust lease is issued, the OS sets a timer that gets fired once T_{out} elapses, causing the trust lease to terminate. With this in mind, two time based terminators can be specified:

- **Timeout:** Provides TrUbi with a relative time value for the termination of the lease. Example: $T_{out} = 1h30m$, meaning that the lease will stay active for 1 hour and 30 minutes since the start of the lease.
- **Date:** Provides TrUbi with an absolute time (day and time of day) to terminate the lease. Example: $T_{day} = 16/Oct - 22:30$, meaning that the lease will stay active until the 16th of October at 22 hours and 30 minutes.

Location Based: Termination rules can also be based on location conditions. These rules confine trust leases to geographical areas defined for example by Global Positioning System (GPS) coordinates or the vicinity to WiFi access points. Two obvious terminators are then envisioned for TrUbi:

- **GPS:** Provides TrUbi with an absolute location (geographic coordinates) and a radius (in meters) within which the lease must be enforced. Example: Terminator $D_{Location=38N,9E} > 10m$ specifies a pivot point, the GPS coordinates, and a maximum radius of 10 meters where the trust lease is valid.
- **Wifi:** Provides TrUbi with a relative location within which the lease will remain active. Essentially TrUbi may receive, through the trust lease specification, the SSID of one, or more, Wifi access points which need to be in range for the lease not to be terminated. Example: $AP_{id=BaseWifi}$ is a

terminator according to which the trust lease will be ended if the device is no longer in range of the access point named ”BaseWifi”.

Explicit Action: An alternative method for terminating a lease is by explicit invocation of `stoplease`. This method aims to allow the strapp that owns the trust lease to terminate it before the termination rules occur. The reason for disabling the lease ahead of time is application specific. To invoke `stoplease` the strapp only needs to provide the trust lease id (l_{id}) as parameter, as follows:

$$\text{stoplease}(l_{id}) \rightarrow \underline{\text{OK}} \mid \underline{\text{FAIL}}$$

By judiciously defining termination rules and invoking `stoplease`, a strapp can generate a rich set of termination events. For example, with this terminator it is possible to imagine an exam scenario where a professor terminates a students exam ahead of time, and correspondent lease, due to catching him cheating off a piece of paper he had brought into the room.

To prevent resources from being indefinitely locked, the OS always sets a ceiling timeout T_{max}. This happens independently of which terminators the trust lease specifies. By doing this users are assured that they will always regain control of their devices.

3.6 Remote Attestation of Trust Leases

For some usage scenarios, strapps may need to convince an external party that the device operates in restricted mode, and possibly even communicate securely with such party. A simple example of this is the scenario of a corporate meeting. Several times the contents of these meetings are confidential and must be kept so. A way to assure such a restriction would be to make meeting attendees to use a strapp for the meeting. This strapp would block both communication and recording channels. However, it is also important for meeting attendees to be sure that their fellow workers are using the strapp and have their devices properly restricted. Taking some ideas from the trusted computing world, this need is addressed by providing a trust lease remote attestation mechanism, which relies on two primitives:

$$\text{quotelease}(n) \rightarrow \underline{\text{q}} \mid \underline{\text{FAIL}}$$

$$\text{verifylease}(q,n) \rightarrow \underline{\text{info}} \mid \underline{\text{FAIL}}$$

Based on these two primitives, the strapp developer can implement a simple protocol between strapp and external party in order to communicate securely and with guarantees that the trust lease is active (see Figure 3.6). Essentially, every TrUbi system has their own asymmetric key pair, properly certified with a digital certificate and authenticating them towards both strapps and third parties. When the communication starts, first the strapp opens a secure channel with the server (e.g., using Secure Sockets Layer (SSL)). Over the secure channel, the server challenges the strapp by sending it a nonce, receiving a signed quote from TrUbi, and checking the quote. If the quote is valid, i.e., contains the right lease

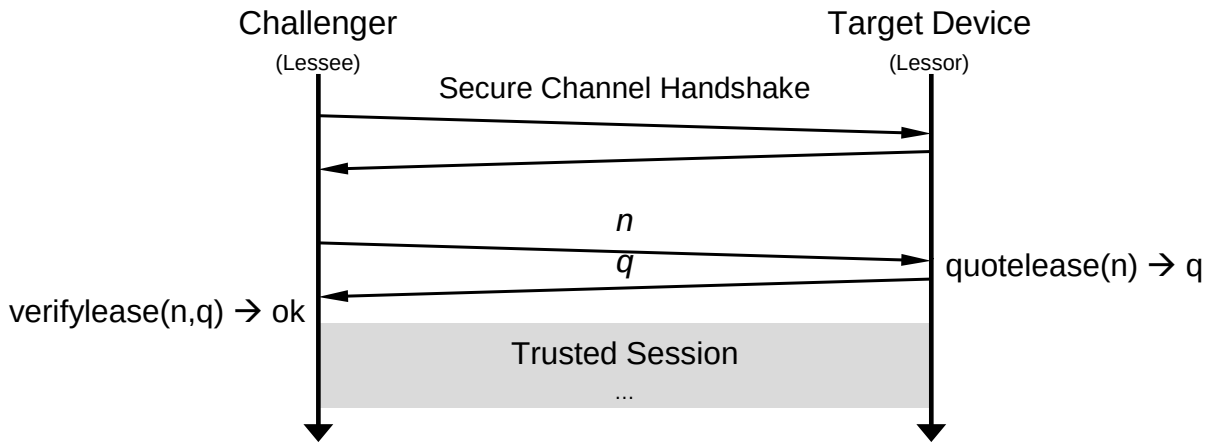


Figure 3.6: Time diagram of trust lease attestation protocol.

description, nonce and TrUbi's signature, the strapp endpoint is trustworthy and the communication is safe. To remotely detect that the trust lease ended, the strapp developer can set a trust lease callback that closes the SSL connection, thus signaling the external party that the trust lease has expired and no more data should be sent. By combining trust lease attestation with SSL, it is therefore possible to create a trusted session between both parties.

Next is explained in more detail the cryptographic operations that enable an external party (challenger) to determine the trust lease state of a target device. First of all it is assumed that each OS is provisioned with a unique keypair and that the private key is securely stored by the OS. It is also assumed the the OS has not been tampered with, thus belonging to our TCB (Trusted Computing Base). The public key is certified by the device manufacturer. The key certificate indicates the version of the OS and whether the OS supports trust leases.

Under these assumptions, the basic protocol is as follows. The challenger sends a nonce n to the target strapp, which invokes `quotelease` passing n as input. The OS simply returns the report message:

$$\text{quote} = \langle m||n\rangle_{K^-}, C(K^+)$$

The quote contains a signature of its current restriction mode m (restricted or unrestricted) concatenated with n. The signature is produced with the target device's private key K^-. The certificate of this key is included in the quote, namely $C(K^+)$. The quote is then returned to the challenger, which invokes `verifylease` to check the quote. The quote is valid if: (i) the nonce in the quote matches n, thereby detecting replay attacks, and (ii) the signature checks against K^+ enclosed in the certificate, which must indicate that the OS supports trust leases verifying that m returned in the quote is meaningful.

However, this basic protocol can be further enhanced. First, the challenger may need to validate the identity of the strapp, and the conditions of the trust lease. To enable this, the quote can include the strapp identity (id_A: name, version, and signature) and details about the trust lease (p_{tl}: restriction

policy and termination conditions):

$$\text{quote} = \langle m||id_A||p_{tl}||n\rangle_K^-, C(K^+)$$

This extra information is returned by `quotelease` and can be validated by the challenger. Second, the challenger may need to determine if the endpoint strapp is indeed the lessee of the lease. To ensure this, `quotelease` only returns a quote to the strapp that currently owns the trust lease, otherwise it sends an error message. Lastly, if there are concerns about the identity of the OS, the protocol can take advantage of trusted computing hardware. In this case, the quote must be extended with an additional signature issued with the platform's identity key. This new signature covers the original nonce n concatenated with a hash of the OS calculated upon boot. This hash enables the challenger to validate the identity of the OS.

3.7 Summary

This chapter presents TrUbi's architecture and the inherent trust lease model behind it. TrUbi's architecture is based on a new concept by the name of Trust Leases. Trust leases are contracts between a lessor, the mobile device user, and a lessee, an interested party. Through trust leases, lessees can request the restriction of certain device functionalities while being assured of their enforcement. TrUbi leverages this concept into creating an OS-independent architecture that introduces two modes of execution: (1) the unrestricted mode, where the mobile device behaves as usual and without any restrictions; (2) the restricted mode, where some of the mobile device's functionalities and resources are disabled or restricted. Restrictions are categorized in three main types: Resource rules, Settings rules and Application rules. Trust Leases have a limited duration. Thus, leases can be terminated based on time or location events, as well as by explicit request of their lessees. Lastly, lessees may request TrUbi for a proper attestation of the restrictions' enforcement. To achieve this, TrUbi uses digital certificates. Digital certificates allow for the use of the asymmetric keys to properly sign and cypher messages between lessor and lessee achieving the desired attestation.

4

Implementation

Contents

4.1 TrUbi's Prototype on Android OS 41
4.2 TrUbi's API 44
4.3 Supported Trust Lease Restrictions 45
4.4 Supported Trust Lease Terminators 49
4.5 Trust Lease Persistence 50
4.6 Summary 51

This chapter focuses upon TrUbi's implementation on the Android OS. This implementation has the goal of verifying the feasibility of the TrUbi system, as well as allowing a proper evaluation of this system, discussed in Chapter 5. Section 4.1 focus on implementation specific decisions that influenced the design of the TrUbi prototype. TrUbi's API is explained in Section 4.2 presenting the system calls of TrUbi that allow strapps to use it, as well as some code listings for a better understanding of its usage. Section 4.3 details which restrictions are implemented on the current version of TrUbi prototype, as well as some interesting facts about their implementation on Android-based systems. Section 4.4 mentions the terminators of trust leases that TrUbi currently supports. Lastly, Section 4.5 entails the peculiar issue of lease persistence where actions such as rebooting or completely powering off the mobile device must not compromise the trust lease being currently enforced.

4.1 TrUbi's Prototype on Android OS

As mentioned before, this Section describes the TrUbi prototype and its implementation on Android. TrUbi's prototype architecture, Figure 4.1, rests upon a standard Android stack, which comprises a modified Linux kernel (bottom layer), the Android framework, which includes the Android runtime, libraries, system services, and system apps (middle layer), and client applications (top layer). TrUbi also adds a set of specific components (shown as dark shaded boxes) which implement the trust lease reference monitor.

TrUbi Manager is the core of the system, a system service that enforces trust lease restrictions, coordinates the trust lease lifecycle, and manages transitions between OS restriction modes. The TrUbi Manager serves trust lease calls issued by strapps, which run on the client application layer alongside regular Android applications. To achieve this, the TrUbi Manager component is filled with the necessary trust lease logic and exposes an API (detailed in Section 4.2) for strapps to be able to use the primitives explained in Chapter 3.

Lease Confirmer is a system app also part of the TrUbi prototype. This app is used by TrUbi upon the request of a lease by a strapp to interface with the user and obtain trust lease authorizations. The Lease Confirmer displays the lease conditions to the user and provides buttons for accepting or declining them. A simple example is shown in Figure 4.2 where this app requests authorization for a lease with: (i) duration of one minute and no location restrictions; (ii) restricted access to the sound of the device, meaning, upon enforcement of the lease it will be put in its minimum setting and may only be altered by an app with the package name of com.trubi.app

Hooks are used by TrUbi Manager to enforce access control restrictions. Spread across the system, these hooks monitor every access to restricted objects allowing TrUbi Manager to perform the corresponding control decisions. Some of these hooks are placed in the Android framework (represented

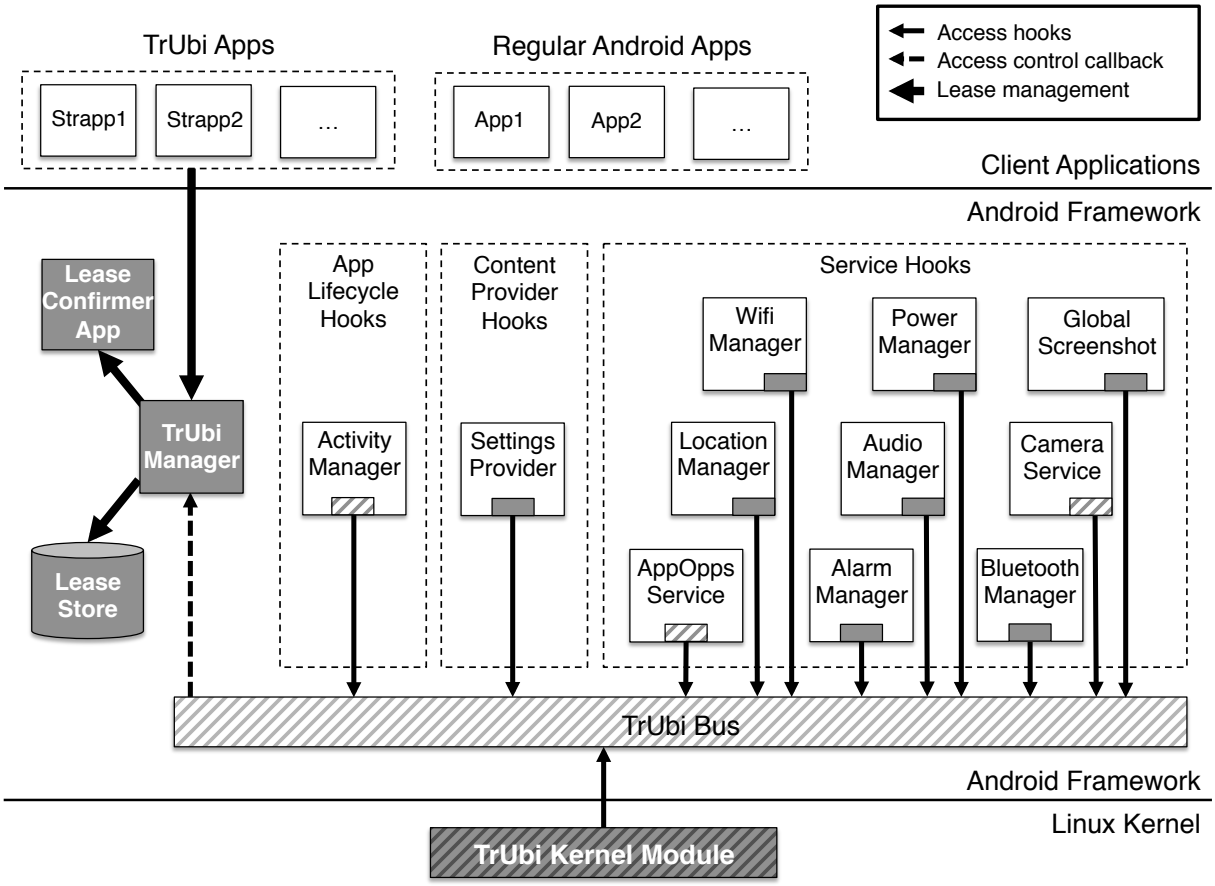


Figure 4.1: Architecture of TrUbi's prototype in Android OS: grey background indicates TrUbi components natively implemented; diagonal stripes under white background means that it was based on an unmodified ASM component; diagonal stripes under grey background denotes that it was modified from a pre-existing ASM component.

in Figure 4.1 as small shared boxes inside pre-existing Android system components). Their goal is to control relevant events involving the lifecycle of applications (e.g., launching an app), access to content providers (e.g., system settings), or access to system services that control certain resources (e.g., sound, or camera). Some other hooks are located in the Linux kernel (e.g., for restricting network access), more specifically in the TrUbi kernel module. These last hooks are specially important to prevent the bypassing of the system restrictions since some resources may be accessed both at the middleware layer and at the kernel. To streamline the communication between hooks and TrUbi Manager, access events are routed from the hooks to the TrUbi Manager through a dedicated system service named TrUbi Bus.

To implement all required hooks, it is necessary to instrument multiple Android software components, both in the framework and kernel. To reduce the complexity of this task, TrUbi leverages ASM [5]. As explained before (Chapter 2), ASM is a security framework that places hooks in the Android system and allows application developers to override the system's default access control mechanisms by imple-

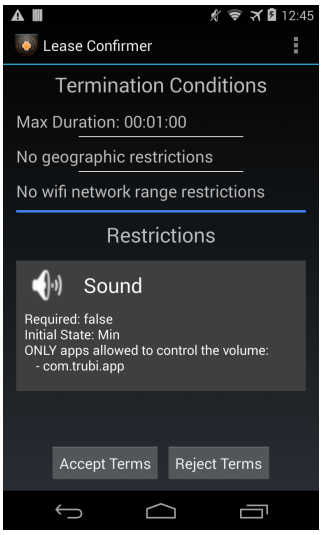


Figure 4.2: Lease Confirmer app.

menting custom access decisions in their application code. As a result, ASM can be used to implement novel access control mechanisms, including TrUbi's. ASM provides a backbone for placing and handling hooks in both the framework and kernel:

- The TrUbi Manager is based on a modified version of an ASM APP that was transformed into a service and filled with trust lease logic.
- The TrUbi Bus is implemented by the ASM Service.
- The TrUbi Kernel Module is based on the ASM LSM for intercepting kernel hooks.
- Some framework hooks are provided by ASM (in Activity Manager and Camera Service)

However, ASM alone is insufficient to implement TrUbi. Various framework hooks were unsupported by ASM and had to be implemented. Among these hooks are the ones for volume control, brightness, microphone, camera LED, airplane mode and others. Modifications were also done to the ASM kernel module (to control suspension and termination of application processes). The remaining TrUbi components were built from scratch.

In this work, TrUbi's prototype is using an ASM implementation from November 2014, which requires Android KitKat. For this reason, the TrUbi prototype was built around the relatively outdated Android KitKat distribution. TrUbi source code was written in Java (10KLOC) and C (0.2KLOC).

```
1  // get reference to the TrUbi Manager
2  IASMAPP trubi = IASMAPP.Stub.asInterface(ServiceManager.getService("TRUBI"));
3
4  // parse lease from XML file on "path"
5  LeaseHandler handler = new LeaseHandler(path);
6  Lease lease = handler.parseLocal();
7  try {
8    // instantiate the trust lease in the system
9    int leaseId = trubi.startLease(lease);
10 } catch (TrUbiRemoteException e) {
11   // lease creation failed
12 }
```

Listing 4.1: Java code fragment to start a lease.

4.2 TrUbi's API

TrUbi provides strapp developers with an API that implements the basic trust lease primitives introduced in Chapter 3: `startlease`, `stoplease` and `quotelease`. The `verifylease` primitive is not part of the API but is instead provided to strapps through a library implemented in TrUbi. However, to be able to implement TrUbi in a timely manner some of these primitives were slightly altered. Bellow is explained every primitive's implementation and what differs from their original architecture of Chapter 3.

To start a trust lease, developers must follow the sequence of steps illustrated in the sample Java code of Listing 4.1: first obtain a reference to the TrUbi Manager, then read the restriction policy from a XML file into a local object instance, and finally invoke the method `startlease` with the restriction policy.

Listing 4.2 shows the XML file of a restriction policy to mute the sound volume for one hour and a half (1h30). The whitelist parameter means that no other application will be allowed to unmute the sound except the mTicket.com strapp (identified by its package name). As mentioned before (Section 3.1), mTicket is an hypothetical strapp for cinema attendees to have their devices muted during the screening of movies. Alternatively to XML files, the restriction policy can be initialized programmatically. To terminate the trust lease explicitly (not shown in Listing 4.1), the developer must invoke `stoplease`. For such an action the procedure is similar to that of the start of the lease but the method that is used is `stoplease`, with no arguments.

The trust lease remote attestation primitives follow the cryptographic protocols specified in Section 3.6. For symmetric cryptography TrUbi prototype uses AES-256, and for digital signatures RSA-1024 and SHA-2. The cryptographic keys are secured in Android's Key Store. In the current implementation the integrity of the kernel is not verified with trusted computing hardware. This is because both the kernel and the operating system are part of the Trusted Computing Base of this project and their integrity is considered to be assured. However, as future work, a proper attestation of the integrity of the kernel and operating system can be achieved through hardware with support for trusted computing primitives.

```
<?xml version='1.0' encoding='UTF-8' ?>
<trustlease>
  <terminator type="Timetout">01:30:00</terminator>
  <restrictor id="Sound">
    <accesslist type="WhiteList">
      <package-name>mticket.com</package-name>
    </accesslist>
  </restrictor>
</trustlease>
```

Listing 4.2: Example of a restriction policy written in XML.

4.3 Supported Trust Lease Restrictions

This section addresses which are the currently supported restrictions of the TrUbi prototype, how they are implemented, and some particularly interesting aspects of their implementation.

Applications Restrictions:

- **Process:** As mentioned before (in Section 3.4), this restriction limits which applications may or may not run during the enforcement of the lease. To achieve this, TrUbi's prototype tackles two main issues: (1) preventing new applications from being launched, (2) suspending already running applications.

  (1) To prevent new applications from starting, TrUbi uses preexisting ASM hooks placed in the Activity Manager. Some of these hooks supervise the lifecycle of applications' main processes and components. TrUbi leverages these hooks to deny operations that result in the execution of new application processes: start Activities and Services, bind to Services, and broadcast intents to apps (which cause the execution of Broadcast Receivers).

  (2) Suspending running applications is not straightforward. In Android, applications may launch background processes. These background processes are not managed by the Activity Manager and therefore can not be directly suspended, or shutdown, through this service. Instead, TrUbi communicates with the Linux Kernel to achieve this goal. When a trust lease starts, TrUbi Manager sends to the TrUbi kernel module a list of processes' UIDs that are allowed to execute (Figure 4.3). From the kernel, TrUbi loops through the system's process list and sends a `SIGSTOP` signal to every process whose UID belongs to the userspace and is not present in the list. When the lease finishes, TrUbi sweeps the process list like before, but this time, sends a `SIGCONT` signal to the previously suspended processes, effectively unfreezing them. To support this (un)freezing operation, TrUbi's prototype extends ASM's Middleware-to-Kernel communication infrastructure.

  Suspending and resuming applications' processes allows TrUbi to cause minimal impact to the

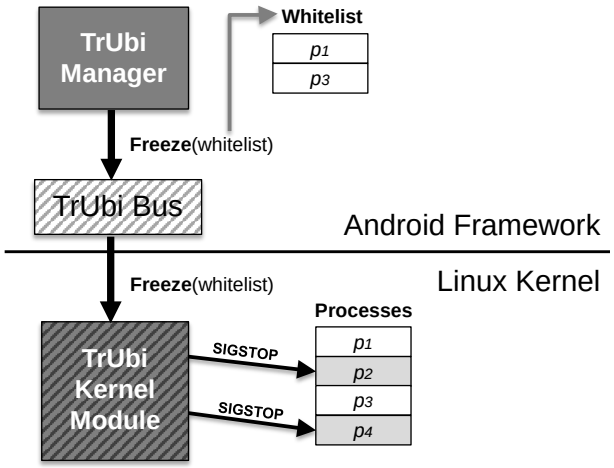


Figure 4.3: Freezing routine of the process restriction.

user's experience. However, if Android begins to experience shortage in resources (e.g., memory), the system may decide to kill suspended processes. To prevent loss of application state, TrUbi fires specific component lifecycle events before suspending an application's processes. Such events will trigger the execution of standard callbacks (e.g., `stopActivity`) so that the application can save relevant volatile state before being terminated.

Resources Rules:

- **Network:** Just like with the process restriction, this restriction also has two main aspects to take into account: (1) on going connections started before the lease, (2) new connections during the enforcement of the lease. To implement this restriction, TrUbi uses two ASM kernel-level hooks: `HOOK_SOCKET_CONNECT` and `HOOK_SOCKET_ACCEPT`. The former is executed every time an application connects on a socket, the latter whenever it receives an incoming connection request. However, since this control is only done on new connections, only those were being restricted by ASM's hooks. To properly enforce already ongoing connections, TrUbi overrides the two mentioned hooks to keep track of all running applications that maintain active network connections. Then, whenever a trust lease starts, TrUbi terminates all applications that have been denied network access, forcing all their network connections to be closed. While the lease is active, socket connect and accept requests will be intercepted by these hooks and validated by the TrUbi Manager. In this prototype, only the socket connect hook was natively provided by ASM. The socket accept hook was provided by the LSM interface.
- **Camera:** Android supports capturing image and video through a system service named Camera Service, which in turn is accessible to applications through the Camera API. This restriction denies access to the camera by all applications unless specified otherwise. To implement it, TrUbi

leverages a pre-existing ASM hook from the Camera Service. This hook is executed the first time an application accesses the camera. Once again, on going uses are left out of the control of TrUbi and so the solution is similar to that of the network restriction. TrUbi overrides the HOOK_CAMERA hook to keep track of applications that have gained prior access to the camera. If TrUbi needs to revoke their permissions when the trust lease starts, TrUbi terminates such applications.

- **Phone Call:** TrUbi leverages an already existing ASM hook for such an enforcement, named HOOK_CALL, this hook allows a proper access control before a call is actually issued. However, ASM provided no mechanism to control ongoing calls. As so, for ongoing calls, and when a lease is activated, TrUbi checks if said call should be allowed or not. If not, TrUbi revokes its privileges by sending an end call event to the TelephonyService. As opposed to network and camera restrictors, it is not necessary to kill applications currently holding the resource. To control phone call reception events, ASM also does not implement any hooks. TrUbi handles such events using a broadcast receiver that listens to incoming calls and rejects them if a lease is currently being enforced.

- **SMS & MMS:** The implementation of this restrictor is straightforward. Since ASM provides hooks that are invoked every time an application sends or receives an SMS / MMS, all of these operations can be intercepted by TrUbi. Therefore, TrUbi only needs to intercept all these events and make the security policy decision indicated by the trust lease.

- **Screenshot:** Android provides three ways for taking screenshots of the device's screen: (1) an API call that applications can use to capture an image of their own screen, (2) the System UI service, which can be activated by the user (e.g., by pressing the power and volume down buttons), and (3) through the Android Debug Bridge (ADB) interface, which allows a debugger to obtain a picture of the screen. TrUbi ignores mechanism 1 and enables trust leases to disable mechanisms 2 and 3. Mechanism 1 is not controlled because applications can only record, on their screenshots, objects which they themselves drawed through Android services. To implement this restrictor, TrUbi has a third party ASM hook in the System UI service which intercepts and blocks accesses denied by the trust lease. To disable the ADB interface, TrUbi controls the execution of screencap, an external program used by Android's ADB server to take screenshots.

- **Microphone:** TrUbi implementation of this restriction is done by placing a new hook in the Audio Manager's function responsible for toggling the microphone mute function. This hook intercepts all requests to unmute the microphone and aborts them if a trust lease is enabled. In the current implementation, this restriction applies to every application in the system, including the strapp that owns the trust lease. Ideally, the access to the microphone should be fine grained. This means that instead of blocking the whole system's access to the microphone, TrUbi would be able to restrict access to only a subset of applications. However, due to time limitations, such an approach could

not be followed through and is therefore something to be looked at as future work.

- **LED:** The LED is typically used as a camera flash light. Given that the LED is also controlled by the Camera Manager, TrUbi implements this restriction by placing a new hook in the Camera Manager's function where the LED mode is changed. A peculiarity of this restriction is that, since the LED is a property of the camera, there are inter-dependencies between LED and camera: if the LED is restricted, the camera can still be accessed, however, if the camera is restricted, the LED setup cannot be changed.

- **Bluetooth:** To implement this restriction, TrUbi placed new hooks in the Bluetooth Manager service which prevent modification of bluetooth configurations by the user through the System Settings or by applications with the required permissions. Given that Bluetooth uses an alternative communication stack to the typical socket interface, restricting the network with the Network restrictor would not prevent communication between nearby devices over Bluetooth. To allow applications to block such communication channels, TrUbi provides the Bluetooth restrictor, whose goal is to issue trust leases that force the Bluetooth interface to be disabled. When the lease is active, this restrictor affects all applications in the system. Once again, just like the microphone restriction, the ideal scenario would have Bluetooth restricted only to specific applications rather than the whole system. Time limitations didn't allow for a proper research to achieve this and as so this restriction is coarse grained. Future work will be done on this issue to allow a fine grain control. Some difficulties are expected among which two stand out: (1) identifying the proper control point, in Android's implementation of the Bluetooth protocol; (2) being able to identify which applications started the communication, thus distinguishing which ones to allow and which to deny.

Settings Rules:

- **Sound:** For the implementation of this restrictor, TrUbi placed new hooks in all entry points to the Audio Manager where the sound volume can be changed. Given that the Audio Manager controls multiple sound streams (alarm volume, ring volume, media volume), the number of entry points is considerably large, requiring the placement of several hooks in different functions (e.g. `forceVolumeControlStream`, `playSoundEffect`, `playSoundEffectVolume`).

- **Brightness:** Typically, the user can change the screen brightness settings through the system Settings application. Applications can also perform this operation so long as they have the required permission. To implement the brightness restriction, TrUbi has three new hooks in the entry points of all relevant system services where the screen settings are controlled: Window Manager, System Settings, and Power Manager.

```
<?xml version='1.0' encoding='UTF-8' ?>
<trustlease>
  <terminator type="GPS">
    <latitude>55.0</latitude>
    <longitude>9.0</longitude>
    <radius>100</radius>
  </terminator>
  <restrictor id="LED">
    <accesslist type="WhiteList">
      <package-name>mticket.com</package-name>
    </accesslist>
  </restrictor>
</trustlease>
```

Listing 4.3: Example of a lease's XML with a GPS terminator.

- **Airplane Mode:** TrUbi provides a specific restrictor for keeping airplane mode on/off until the trust lease terminates. Since airplane mode can only be toggled by the System Settings (since Android 4.2 onwards), it was only necessary to place a third party ASM hook in the function responsible for the mode switch: `putStringforUser`.

- **Time:** TrUbi implements this restriction through new hooks that were placed in two system services: System Clock and Alarm Manager. These services are reached through the System Settings, which is the only application that is authorized to modify the time settings of the system.

4.4 Supported Trust Lease Terminators

TrUbi's implementation on Android supports the previously mentioned terminators of Section 3.5. We now explain how each one of these is implemented.

Time Based Terminators

- **Timeout:** When TrUbi starts a lease with the time based terminator it launches a thread that sleeps for the duration of the lease; upon waking up, it terminates the lease.

- **Date:** TrUbi uses the system time and absolute date from the lease specification to calculate a timeout and proceeds as if it had been given a `timeout` terminator.

Note: When both terminators are applied in a single lease, TrUbi calculates the shortest of both and only launches one thread for that one.

Location Based Terminators

```
<?xml version='1.0' encoding='UTF-8' ?>
<trustlease>
  <terminator type="Wifi">
    <wifi_networks>trubi</wifi_networks>
    <all_wifi_networks>true</all_wifi_networks>
  </terminator>
  <restrictor id="Process">
    <accesslist type="WhiteList">
      <package-name>mwork.com</package-name>
    </accesslist>
  </restrictor>
</trustlease>
```

Listing 4.4: XML of a lease with a Wifi terminator

- **GPS:** Leveraging ASM's location hooks, whenever the mobile device's location changes, TrUbi verifies if the mobile device is still within the lease enforcement range. In case it is, nothing happens, in case it has left the enforcement range, then TrUbi terminates the lease. Lease example in Listing 4.3.

- **Wifi:** Upon the enforcement of the lease, TrUbi launches a thread that checks the nearby Wifi access points every once in a while. It then checks if the specified access points of the lease are in range and proceeds accordingly. Listing 4.4 is an example of a trust lease with a Wifi terminator.

4.5 Trust Lease Persistence

The TrUbi Manager is responsible for keeping track of active trust leases. However, keeping this information in volatile memory alone opens some vulnerabilities. In particular, if the device is rebooted, information of active trust leases would be lost, resulting in a potential violation of their assurances. TrUbi's prototype addresses this problem by backing up all relevant trust lease state data persistently. Essentially, as soon as TrUbi receives a lease request, followed by the corresponding user approval, it writes into persistent memory the whole lease specification. On every single boot, TrUbi always checks first for any lease specifications that it might have stored in a previous session. This way, if for some unforeseen reason the mobile device reboots, either due to technical reasons, or malicious ones, the lease is always assured to be properly enforced. Upon the termination of the lease, TrUbi deletes from storage the correspondent lease specification.

4.6 Summary

This chapter explains TrUbi's implementation on top of the Android Operation System. TrUbi modifies both the Linux Kernel and the Android Framework, i.e., both bottom and middle layers of Android. TrUbi's main components are: TrUbi Manager, Lease Confirmer and Hooks. TrUbi Manager is the core of the system which enforces the trust lease restrictions, the Lease Confirmer allows TrUbi to ask the user's approval on a lease request, and finally, the hooks are the control points placed on resources and settings which allow TrUbi to control potential accesses to them. To achieve a timely implementation, TrUbi leverages ASM's hooks system and extends it. TrUbi also exposes an API in order to allow strapps to communicate with it. This API allows for a request of a lease, `startlease`, termination of one, `stoplease` and request of a quote of the current lease, `quotelease`. TrUbi currently supports 13 restrictions, where 11 out of 13 are at a fine grain level, meaning, they can restrict subsets of applications instead of the whole system. The remaining two restrictions, namely microphone and bluetooth, restrict the whole system. TrUbi also allows for both time and location terminators, respectively, Timeout/Date and GPS/Wifi. Lastly, TrUbi also tackles the issue of lease persistence. To achieve this, TrUbi persistently stores its currently being enforced lease and checks said store on boot, loading any lease that might have been stored there.

5

Evaluation

Contents

5.1 Restrictions Enforcement Efficiency 55
5.2 System Overall Performance 55
5.3 Trust Lease Primitives Performance 57
5.4 Energy Consumption 60
5.5 Developer Friendly 61
5.6 Summary 63

This chapter focuses on the evaluation of the TrUbi prototype. As mentioned before, TrUbi's prototype is required to be efficient and secure. In this chapter TrUbi is measured in terms of security efficiency, proving its viability in applying security policies, and in performance and energy consumption efficiency, proving the fulfillment of its efficiency requirement. First, Section 5.1 presents the enforcement efficiency of TrUbi's prototype restrictions. Section 5.2 then presents an overall benchmark of the system where generic functions such as graphics processing, CPU, memory and filesystem are compared between the unpatched Android 4.4.1 Android Open Source Project (AOSP) build and the TrUbi patched version. The performance of the primitives for trust leases, namely: `startlease`, `stoplease`, `quotelease` and `verifylease` is measured and presented in Section 5.3. In Section 5.4 TrUbi's prototype is also compared to the original Android 4.4.1 AOSP build in terms of energy consumption. Lastly ASM framework is compared with TrUbi in terms of developer friendliness which was also one of this projects requirements.

Hardware Testbed: Throughout the benchmarks present in this chapter, TrUbi's implementation was tested in a Nexus 4 smartphone. This device features a quad-core CPU of 1.5 GHz, 2GB of RAM, 16 GB of memory with no SD-Card support, 802.11 WiFi interface, 768 x 1280 display, an 8 MP camera with 3264 x 2448 pixels and a LED flash. The running operating system is an Android 4.4.1 AOSP build.

5.1 Restrictions Enforcement Efficiency

TrUbi, as a security framework, has its main concern on security. For this reason the first major concern is to test each and every implemented restriction to prove that it is working as intended. With this goal in mind, we use 10 representative applications for each of the supported restrictions. Table 5.1 has all the implemented restrictions and the corresponding applications used to test them. The applications were chosen based on the resources they use and on their ratting on Google Play [20]. However, the screenshot, airplane mode and time restrictions could not be tested with 10 different applications. This is due to the fact that these resources and settings have special ways of being used and not every application can have access to them. Nevertheless, in all cases, every restriction was successfully tested and properly blocked.

5.2 System Overall Performance

This section evaluates the overall impact of TrUbi's implementation. The software used for this benchmark is an Android application named Softweg [21]. Softweg was chosen because it was used to benchmark other security systems such as SE Android [11]. This application measures the throughput of graphics, CPU, memory and filesystem. The measurements are taken while stress testing each of these components several times (the number of times is user configurable). For a proper evaluation of

Restrictions	Test Applications
Process	Ted (2.3.6), SpeedTest (3.2.8), Spotify (2.6.0.813), Instagram (6.20.2), Color Flashlight (3.6.8), VC Roys (1.6.0), Textra SMS (2.32), ExDialer (191), VR Iovekara (1.2.28), System Camera
Network	Chrome (34.0.1847.114), Facebook Messenger (27.0.0.48.14), Facebook (32.0.0.23.15), Youtube (10.18.55), Instagram (6.20.2), Skype (5.3.0.65524), Spotify (2.6.0.813), TeamViewer (10.0.2484), Ted (2.3.6), SpeedTest (3.2.8)
Camera	Google Camera (2.4.025), Camera HDR Studio (1.2.2), Geak Camera (0.29 beta), Open Camera (1.25), Camera for Android (2.2), HD Camera Ultra (2.2.2), Retrica (2.4.2), HD Camera (4.4.2.3), High-Speed Camera (3.1.1), System Camera
Phone Call	VR Splend (1.3.6), ZERO Dialer (1.01), Simpler Dialer (4.0), Truedialer (2.12), PP (3.9), True Phone Dialer (1.3.7), ExDialer (191), Android L UI (4.0), RocketDial (3.8.6.1), Dialer One (2.2.0), System Caller
SMS & MMS	Handcent SMS (6.5.5), GO SMS Pro (6.24), ZERO SMS (1.2), EvolveSMS (3.2.1), Easy SMS (3.6.0), hello sms (2.2.8), SMS KK (1.91), chomp SMS (6.46), Textra SMS (2.32), System Messaging
Screenshot*	Power Button & Decrease Volume Button Down, Android Debug Bridge (ADB)
Microphone	Smart Voice Recorder (1.7.1), Voice changer (3.1.10), VR Iovekara (1.2.28), ASR (36), Audio Recorder (1.00.28), VR Rabbit (1.6.0), Easy Voice Recorder (1.8.1), VR Korrisoft (1.9), QuickRec (2.5.1b2), VR Splend (1.3.6)
LED	Flashlight HD LED (1.79), DU Flashlight (1.08), Brightest LED Flashlight (1.2.5), Flashlight - Torch LED Light (1.8.6), Super-Bright LED Flashlight (1.0.6), Tiny Flashlight (5.2.4), Flashlight LED Genius (1.10.4), LED Flashlight - Torch (1.5), Color Flashlight (3.6.8), Super-Bright LED Flashlight (1.0.6)
Bluetooth	Bluetooth OnOff Toggle Widget (2.1), Bluetooth Switch and Mute (1.0.3), Quick Bluetooth Change (1.1.0), Bluetooth On/Off Widget (1.5), One Tap Bluetooth Manager (1.0), Bluetooth Flick (1.4), Bluetooth Toggle Widget (1.0), Bluetooth Widget (2.0), Bluetooth On/Off (1.0.1), Bluetooth Status Bar Switch (1.3)
Sound	Volume Controller (1.0), Volume Control + (1.44), Volume Butler (1.8.4), Volume Control Plus (3.3), Master Control Volume (1.2), VC Clever (2.9), VC Simple (1.21), VC Firestorm (3.0.4), VC Roys (1.6.0), VC Widget (2.2.1)
Brightness	Screen Brightness Tool (1.3), Well Brightness (1.0), Brightness Level (2.2.0), Display brightness (4.7), Brightness Level Disc (1.8.2), ABC Brightness (3.8), Yet Another Auto Brightness (1.10.1), LED Flashlight - Brightness Toggle Widget (1.0.4), Status Brightness Free (2.8), Brightness Easy (2.0c)
Airplane Mode*	System Settings Application
Time*	System Settings Application

Legend: * Restrictions which can only be tested through specific methods and applications. More details in Section 5.1.

Table 5.1: Applications used to test the prototype restrictions enforcement efficiency.

the prototype, this software was executed 6 times, with 200 readings per resource. The value of 200 readings was also taken from SE Android benchmarks. However, 200 readings gave us no indication of average and standard deviation so these readings were repeated allowing us to determine these measurements. In terms of methodology the device was first flashed with a clean setup of the OS being tested. The executions were then taken in succession of each other while the device was always connected to the power plug to prevent performance fluctuations due to power management settings. Table 5.2 illustrates the readings provided by the benchmark.

By analyzing Table 5.2 it is possible to verify that the impact of TrUbi, when not enforcing any leases, is low and well within the standard deviation values. This happens because most of TrUbi's hooks are not related with the type of operations used in these system performance benchmarks. TrUbi has low impact on GPU and CPU intensive tasks, as well as memory and filesystem. Instead, TrUbi focuses more on resources and settings that are usually accessed through user interaction. This makes overheads ir the order of just a few milliseconds negligible when compared to the average user reaction time. Furthermore, camera, microphone and sound are not usual targets for performance benchmarks. Resources like these are usually benchmarked in terms of quality, i.e., image quality or sound sharpness.

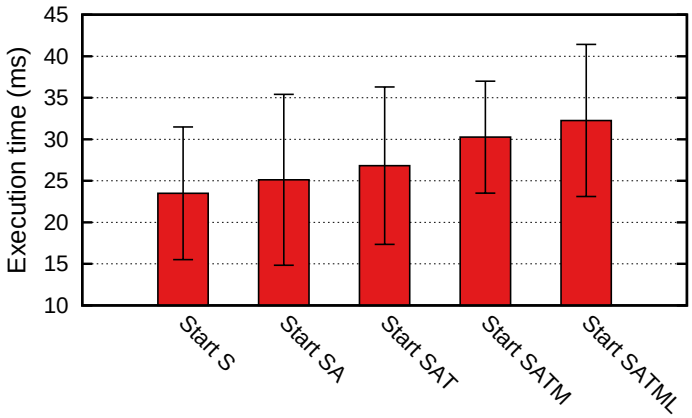
Resources\OS	AOSP	TrUbi
Graphics (Softweg)		
Draw opacity bitmap (MPix per sec)	6.7 $\pm$ 0.03	6.7 $\pm$ 0.01
Draw transparent bitmap (MPix per sec)	6.2 $\pm$ 0.01	6.2 $\pm$ 0.02
CPU Whetstone (Softweg)		
MWIPS DP (DP)	296.2 $\pm$ 2.84	298.0 $\pm$ 2.78
MWIPS SP (SP)	317.5 $\pm$ 1.51	312.5 $\pm$ 2.71
MFLOPS DP (SP)	83.7 $\pm$ 0.49	86.1 $\pm$ 0.57
MFLOPS SP (SP)	80.8 $\pm$ 0.21	82.0 $\pm$ 0.29
VAX MIPS DP (DP)	236.9 $\pm$ 0.84	235.6 $\pm$ 1.51
VAX MIPS SP (SP)	244.9 $\pm$ 1.11	236.0 $\pm$ 1.37
Memory (Softweg)		
Copy memory (Mb/sec)	2201.3 $\pm$ 43.21	2229.8 $\pm$ 20.45
Filesystem (Softweg)		
Creating 1000 empty files (sec)	0.3 $\pm$ 0.01	0.3 $\pm$ 0.01
Deleting 1000 empty files (sec)	0.2 $\pm$ 0.01	0.2 $\pm$ 0.01
Write 1M into file (M/sec)	111.6 $\pm$ 0.39	112.4 $\pm$ 0.46
Read 1M from file (sM/sec)	512.4 $\pm$ 2.08	505.0 $\pm$ 1.62

Table 5.2: Results of Softweg experiments.

5.3 Trust Lease Primitives Performance

This section focuses on measuring the elapsed time for each of the trust lease primitives that the prototype implements. Besides analyzing all four primitives, different scenarios are also taken into account within the same primitive. For example, not only is it important to measure the time of `startlease`, but also how it varies with different restrictions being applied. All the results presented in this section were taken with a clean setup of the TrUbi system on top of the mobile device. The device was then set up with the required applications for the test and none else. Furthermore, each of the experiments was done 50 times and measured for average values and standard deviation. The experiments were done 50 times because it is considered a big enough value to achieve reliable results.

- **Start Lease - No Action:** Figure 5.1 shows the execution times of `startlease` with restrictions: sound, airplane mode, time, SMS & MMS and LED. These restrictions have the particular characteristic that they can be issued without requiring a state change of the system's resources or settings. For this reason they register the lowest measured values for `startlease`, namely $23.5ms$. This value is negligible when compared to a mobile application which takes at least 1 minute to run. It is also possible to verify that `startlease` has a fixed overhead of about $21ms$, which increases, on average, around $2ms$ per restriction added. The fixed overhead of $21ms$ is due to the fact that every time `startlease` is called, independently of the amount of restrictions, some actions must always be performed. Among those we have lease parsing, the terminator thread setup and the persistent lease storage in the mobile device. Another important aspect of these measurements is the standard deviation of $8.73ms$. Such a high value for the standard deviation was traced back to Android's garbage collector. We noticed that every time the garbage collector performed an action



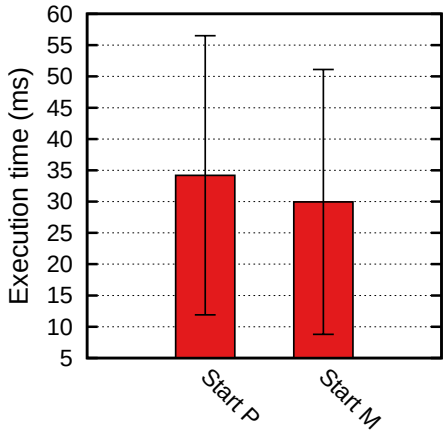
Legend: S: Sound; A: Airplane Mode; T: Time; M: SMS & MMS; L: LED.

Figure 5.1: Execution time of `startlease` with restrictions which require no OS action.

the registered values would spike. Unfortunately, it wasn't possible to know for sure how much overhead each garbage collector action had on said spikes so the values couldn't be corrected. Instead, the most reasonable approach was to present the values as registered and explain them.

- **Start Lease - Simple Action:** Figure 5.2 shows the times of `startlease` for the restrictions microphone and phone call, respectively, $29.94ms$ and $34.2ms$. Once again these values are insignificant comparing to the average execution time of an application. These two restrictions require the OS to take some actions before they are actually being enforced. The microphone restriction requires TrUbi to invoke the `muteMicrophone` method from the AudioSystem service of Android. The phone call restriction requires a similar action so that it can terminate ongoing calls before the enforcement of the lease. However, this restriction also requires the registration of a broadcast receiver to control all incoming calls for the duration of the lease. This last action explains why the phone call restrictions registers slightly higher values for lease activation. Once again the standard deviation registers high values of around $21.73ms$, the reason is the same as before, Android's garbage collector.

- **Start Lease - Application Killing:** Some leases require TrUbi to terminate running applications. This happens when a lease with the network or camera restrictions is issued. For this benchmark TrUbi prototype was tested with scenarios of 1, 5 and 10 running camera applications, respectively K1, K5 and K10 (Figure 5.3). The values of 1, 5 and 10 were chosen to allow us to understand if the time increased linearly as the number of applications terminated did. Registered values were $34.32ms$, $91.46ms$ and $146.3ms$. If we deduct the average overhead of $21ms$, we can conclude that the `startlease` time does indeed go up in a linear way, with an average of $13.75ms$ per application terminated. The value of $146ms$ is already a considerably high value when comparing to the



Legend: P: Phone Call; M: Microphone.

Figure 5.2: Execution time of `startlease` with restrictions which require simple actions from the OS.

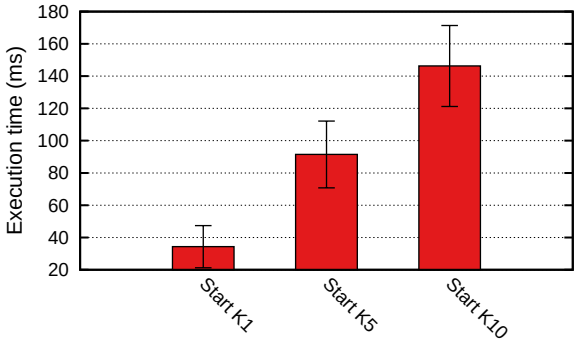


Figure 5.3: Execution times of `startlease` with only one resource, camera. Scenarios with 1, 5 and 10 applications using this resource, respectively K1, K5 and K10.

remaining lease startup times. However, it is still negligible in the overall scheme of applications. Furthermore, it is important to understand that terminating 10 applications is also a considerably large amount of applications as well. Most common uses of mobile devices don't have 10 running applications using the camera, or even running applications with network sockets open.

- **Start Lease - Application Freezing:** The process restriction, as mentioned before, allows to limit the execution of applications. In order to test this mechanism the prototype was measured issuing a lease with such a restriction. The measured values had an average of $41.18ms$ with a standard deviation of $22.89ms$. Android's garbage collector also caused spikes to happen on these tests. Because the applications were being frozen, sometimes Android's garbage collector would perform a cleanup of their resources. This made the micro-benchmark to register big standard deviation values. Note that this value includes freezing every non-essential Android application and service. The registered value is thus considered the lowest value one can achieve with such a lease.

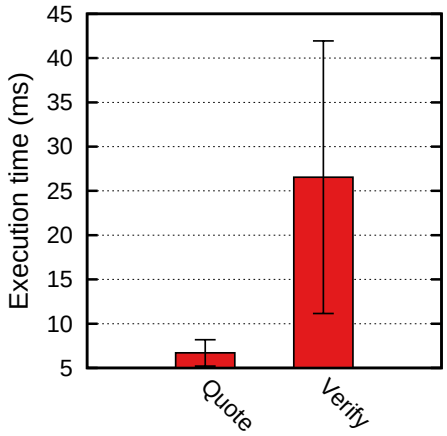


Figure 5.4: Execution times of `quotelease` and `verifylease`.

Benchmark	AOSP	TrUbi	Overhead
Consumption (J)	2207.8 $\pm$ 90.3	2218.8 $\pm$ 123.7	0.5 %

Table 5.3: Energy consumption overhead of TrUbi prototype.

- **Stop Lease:** Terminating a lease is simple and quite straightforward. As so, the time registered for this trust lease primitive during these benchmarks was the lowest. The `stoplease` primitive was executed with an average time of $9.66ms$ and a high value of standard deviation of $7.04ms$, once again due to Android's garbage collector cleanup actions.

- **Quote & Verify Lease:** The `quotelease` and `verifylease` are the two last benchmarks present in this section. In Figure 5.4 is noticeable that there is a big discrepancy between these two primitives. The measured values were $6.7ms$ and $26,54ms$, respectively `quotelease` and `verifylease`. This is due to the fact that `quotelease` performs a single digital signature, while `verifylease` has two signatures to verify and one hash: from the quote and the certificate (see Section 3.6).

5.4 Energy Consumption

This section compares the energy consumption of TrUbi's prototype and the unpatched version of Android 4.4.1 AOSP. The benchmark was done using a software named Trepn profiler 6.0 [22] by Qualcomm. Trepn obtains power consumption samples every $100ms$. It can report accurate values by accessing the power management integrated circuit available in some devices, including the testbed's device: Nexus 4. This benchmark was done in the same test environment as that of the performance benchmark, only difference being that the device is no longer connected to the charger. The test was done running the Trepn software 10 times while it evaluated the Softweg benchmark which was configured to do 50 repetitions per resource. Running Trepn while performing a Softweg benchmark allowed

```
1  public class ASMRestrictCameraApp extends IASMAPP.Stub {
2
3    //Return values to allow or deny the access verification
4    private static final int ASM_ALLOW = 0;
5    private static final int ASM_DENY = -1;
6
7    //Parsing the allowed camera users from a file to a List
8    private static final List<String> allowedCameraUsers =
9                    parseCameraUsersFromFile("allowedCameraUsers.txt");
10
11   //Registering ASM hooks necessary to enforce the desired policy
12   public String[] initialHookList() throws RemoteException {
13     return new String[] {
14       // Camera Hook
15       Hook.HOOK_CAMERA,
16     };
17   }
18
19   //Callback function for access control purposes
20   public int camera(int uid, String packageName) throws RemoteException {
21     if(allowedCameraUsers.contains(packageName)){
22       return ASM_ALLOW;
23     }else{
24       return ASM_DENY;
25     }
26   }
27 }
```

Listing 5.1: Java code of a simple ASM application to restrict the camera usage.

both systems to be stress tested in the same testing environments achieving accurate results. The fact that Softweg was configured to do 50 repetitions per resource provided a lengthy stress test so that Trepn could take several measurements. Furthermore, repeating this process 10 times allowed to compensate any system performance fluctuations, thus registering average and standard deviation values. Table 5.3 shows a slight increase in power consumption by 0.5%, which is well within the standard deviation. The slight increase in power consumption of 0.5% was already expected as TrUbi's overall performance is just slightly inferior than that of the original Android AOSP version.

5.5 Developer Friendly

Developer friendliness is another of TrUbi's requirements. The initial plan was to have android application developers go through the process of developing security sensitive applications with and without TrUbi's framework. However, due to time constraints this was not possible. Instead, we present here some code listings to compare the complexity of implementing the same functionality using both ASM and TrUbi.

The comparison is done between ASM and TrUbi because these two systems are conceptually close,

```
1  public class Strapp extends Activity {
2
3    private void startCameraEnforcement() {
4
5      // get reference to the TrUbi Manager
6      IASMAPP trubi = IASMAPP.Stub.asInterface(
7                                        ServiceManager.getService("TRUBI"));
8
9      // parse lease from XML file on "path"
10     LeaseHandler handler = new LeaseHandler("lease.xml");
11     Lease lease = handler.parseLocal();
12
13     try {
14       // instantiate the trust lease in the system
15       int leaseId = trubi.startLease(lease);
16     } catch (TrUbiRemoteException e) {
17       // lease creation failed
18     }
19   }
20 }
```

Listing 5.2: Java code of a simple strapp

i.e., both envision their usage by application developers to develop safer systems and applications. Since Android itself does not allow applications to restrict the mobile device's resources, comparing code between Android and TrUbi is futile.

Listing 5.1 shows the required code to develop a simple ASM application which is able to restrict the access to the camera of a mobile device. The developer must write an application which extends from the IASMAPP interface. Then it must be aware that 0 and -1, respective, represent `allow` and `deny`, in terms of access control. Afterwards the developer must register for the required hooks, offered by ASM, to achieve the desired enforcement, `HOOK_CAMERA` in this case. Last, the developer must override the callback function and program its access control logic based on the arguments of this function and on the applications he desires to allow such an access. When compared to Listing 5.2, it is possible to understand that TrUbi requires less lines of code. It is also possible to understand that TrUbi has three main steps that are always the same, independently of restrictions being applied: (1) acquisition of the TrUbi service; (2) parsing of the lease XML file; (3) issuing of the lease through the `startLease` method.

To better understand the difference between these two security frameworks, Listing 5.4 presents the simplified code required by an ASM application which goal is to stop unauthorized applications from running. Since Android allows applications to start from several different scenarios, an application developer must first of all be aware of all of them. Afterwards, application developers need to register hooks to all these events and override their callback functions. This requires application developers to both be aware of how Android is built, and at the same time be aware of security vulnerabilities of the operating system. Contrasting with this, the previously shown Listing 5.2 of a TrUbi strapp can also implement a strapp to restrict applications from running. The issuing of trust leases by strapps is always

```
<?xml version='1.0' encoding='UTF-8' ?>
<trustlease>
  <terminator type="Timetout">01:30:00</terminator>
  <restrictor id="Camera">  <restrictor id="Process">
    <accesslist type="WhiteList">
      <package-name>allowApp1.com</package-name>
      <package-name>allowApp2.com</package-name>
      <package-name>allowApp3.com</package-name>
    </accesslist>
  </restrictor>
</trustlease>
```

Listing 5.3: XML file of a lease that restricts the camera (black) or applications (red)

the same and independent of what they wish to restrict. The only change required is on the lease's XML file. Nevertheless, even said change in the XML file is minimum and is exemplified in red in Listing 5.3.

After analyzing the above stated code it is possible to conclude that TrUbi fulfills its requirement of developer friendliness. Through the use of TrUbi's security framework it is possible for application developers to specify their security needs in a quick way and with little effort. The framework also allows developers to state said security needs without requiring them to know implementation specific details.

5.6 Summary

This chapter benchmarks TrUbi's prototype. The benchmarks are done on a Nexus 4 smartphone with either TrUbi's implementation or Android's 4.4.1 AOSP build. First we prove TrUbi's security efficiency by running 10 representative applications per supported restriction of TrUbi's prototype. In all cases the tests were successful and the resource, or setting, properly restricted. Next followed an overall performance benchmark. Through the use of a software called Softweg, both TrUbi and Android 4.4.1 were stress tested. The results show that both systems have a similar performance and that TrUbi's impact on the overall system is negligible. TrUbi's implementation of the trust lease primitives was also tested. Throughout several tests it is possible to understand that even the simplest lease has a fixed overhead of around $21ms$. This overhead is expected as every lease must be parsed, properly stored in persistent memory and its terminator thread set. More complex leases, which require the termination of processes or their suspension, were also tested. These leases take longer to activate. However, the highest value registered was of $146.3ms$, for a lease that required 10 applications to be terminated. TrUbi prototype is also measured in terms of energy consumption efficiency, scoring a 0.5% overhead when compared to the unpatched version of Android 4.4.1. Lastly, TrUbi shows its developer friendliness by comparison to ASM.

```
1  public class TrubiManagerService extends IASMAPP.Stub {
2    private static final int ASM_ALLOW = 0;
3    private static final int ASM_DENY = -1;
4
5    //Parsing the allowed camera users from a file to a List
6    private static final List<String> allowedApplications =
         parseAllowedApplicationsFromFile("allowedApplications.txt");
7
8    public String[] initialHookList() throws RemoteException {
9      return new String[]{
10       // ASM Hooks
11       Hook.HOOK_BROADCAST_INTENT_MOD, Hook.HOOK_START_ACTIVITY,
12       Hook.HOOK_START_ACTIVITY_MOD, Hook.HOOK_START_SERVICE,
13       Hook.HOOK_START_SERVICE_MOD, Hook.HOOK_BIND_SERVICE,
14       Hook.HOOK_BIND_SERVICE_MOD };
15   }
16
17   public String[] resolveBroadcastReceivers_mod(List<ResolveInfo>
         resolvedList,...) throws RemoteException{
18
19     ArrayList<String> list = new ArrayList<String>();
20
21     for (ResolveInfo info : resolvedList) {
22       if (allowedApplications.contains(info.activityInfo.packageName)) {
23         list.add(info.activityInfo.packageName); }}
24     return ArrayListToString(list); }
25
26   public int startActivity_mod(..., ActivityInfo aInfo,...) throws
         RemoteException{
27     if (allowedApplications.contains(aInfo.packageName)) {
28       return ASM_ALLOW; } else { return ASM_DENY; }
29   }
30
31   public int startActivity(..., ActivityInfo aInfo,...) throws
         RemoteException{
32     if (allowedApplications.contains(aInfo.packageName)) {
33       return ASM_ALLOW; } else { return ASM_DENY; }
34   }
35
36   public int startService(...) throws RemoteException{
37     ...
38   }
39
40   public int startService_mod(...) throws RemoteException{
41     ...
42   }
43
44   public int bindService(...) throws RemoteException {
45     ...
46   }
47
48   public int bindService_mod(...) throws RemoteException {
49     ...
50   }
51 }
```

Listing 5.4: Simplified code of an ASM application that restricts applications from running.

6

Conclusion

Contents

6.1 Conclusion 67

6.2 Future Work 68

6.1 Conclusion

Mobile devices are highly integrated into today's society lifestyle. Due to a large set of functionalities, these devices are useful in a wide variety of scenarios. However, this large set of functionalities is also a security liability in some usage scenarios. Certain scenarios require that mobile devices have their functionalities restricted or even disabled. When in a cinema room, mobile devices should have their volume locked in the minimum setting. In a more security sensitive scenario, like a private corporate meeting, attendees should be assured that no device in that room can record what is being discussed. Other examples can be pointed out, among them, examinations done through the use of Tablets, the acquisition of medical records which must not be of public knowledge, online voting for presidential elections and other official matters. All these scenarios leverage mobile devices mobility and functionalities but are also compromised by them.

This project introduces a new security framework by the name of TrUbi which enables a new mode of operation named restricted mode. This mode allows for the restriction of certain functionalities of the mobile device, for a limited amount of time. Restrictions are categorized in three sets: (1) applications, restricting which applications may or may not run; (2) resources, restricting access to specific mobile resources like network, bluetooth, microphone, camera and others; (3) settings, restricting the configuration of system settings such as volume, brightness, airplane mode and time. TrUbi, following an application-centric approach, allows for strapps, trusted applications which are aware of the existence of TrUbi, to request the enforcement of security policies which in turn apply these restrictions. If the terms of such policies are accepted, by the user of the mobile device, then a trust lease [7] is issued upon the operating system. Trust leases are essentially a trust contract between a lessor (the mobile device) and a lessee (an interested party, internal or external). While under a trust lease, the mobile device has access to a very concrete set of functionalities. At the same time, the lessee may request TrUbi to assure him that the restrictions are in place. This is done through the usage of specific remote attestation methods, namely `quotelease` and `verifylease`.

TrUbi implementation was done on top of Android 4.4.1, commonly known as Android Kitkat. TrUbi prototype is composed by: (1) the TrUbi Manager, a core system service which enforces the trust leases restrictions and exposes an API to be used by strapps; (2) the Lease Confirmer, a system application responsible for delivering the lease specification to the user and requesting its approval. (3) the Hooks, callback routines spread along the middleware and kernel of the system which allow the enforcement of the access control restrictions. For a timely implementation, TrUbi leveraged Android Security Modules (ASM) [5]. As explained in Chapter 2, ASM is a security framework that places hooks in the Android system and allows application developers to override the system's default access control mechanisms.

Finally, the TrUbi prototype was benchmarked in different aspects for a proper assessment of its viability. First, TrUbi was tested with a set of applications. This was done in order to ensure that every

implemented restriction was properly being enforced, which they are. Doing so allowed to assess the efficiency of TrUbi's security measures on the system. Next, TrUbi's performance was tested through a general and overall benchmark using a software named Softweg [21]. This software stress tests the CPU, GPU, memory and filesystem, ratting the system accordingly. The results registered showed that TrUbi is very close to the original unpatched version of Android Kitkat. The most extensive testing of the prototype was done on the API and consequent system execution of such functions. The evaluation of the API focused on benchmarking the average execution time for the trust lease primitives, namely: `startlease`, `stoplease`, `quotelease` and `verifylease`. Through these tests it was possible to understand that the average time for a simple, one resource, trust lease is of $23.5ms$, a negligible amount when compared to an application which takes more than one minute to run. This value goes up an average of $2.19ms$ per resource added to the lease specification. Lastly, TrUbi's energy consumption was also benchmarked, registering an overhead of 0.5% according to the Trepn profiler 6.0 [22] software.

This allows this project to be concluded in a positive remark. The goals of the project are fulfilled by both TrUbi and its implemented prototype on Android 4.4.1. In short, TrUbi trust lease based design allows for flexible security policies and developer friendliness. This is done by enabling security policies to target specific resources and subjects. These targets are described in the lease's specification by application developers with ease. TrUbi also allows for internal and external assurances through the attestation methods previously explained in Section 3.6. The efficiency and viability of TrUbi was proven through the benchmarks done and presented in Chapter 5.

6.2 Future Work

The TrUbi prototype is at the moment a functional prototype. However, some interesting aspects still need to be studied and polished in order to achieve a proper public release. With this in mind, in the future we will keep on developing and enhancing TrUbi with the goal of releasing the TrUbi to the scientific community. Among the several aspects that are to be worked on, the most important ones are: (1) researching the microphone and bluetooth routines on Android OS, allowing us to turn their restrictions into fine-grained restrictions; (2) implementing a callback function to be called upon the termination of the lease, like the one described in Section 3.3; (3) writing detailed documentation of the TrUbi architecture and implementation, for the scientific community; (4) developing an "Hello World" strapp, with proper documentation, for application developers. Furthermore, there are also more complex enhancements that can still be done to the TrUbi project. The most compelling among these being the reduction of the TCB through the use of trusted computing primitives and hardware, which can be the subject of a future Master's Thesis due to its complexity.

Bibliography

[1] M. Conti, V. T. N. Nguyen, and B. Crispo, "CRePE: Context-Related Policy Enforcement for Android," *Information Security*, vol. 6531, pp. 331–345, 2011.

[2] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin, "Permission Re-Delegation: Attacks and Defenses," in *Proc. of USENIX Security*, 2011.

[3] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel, "Semantically Rich Application-Centric Security in Android," *Security and Communication Networks*, vol. 5, no. 6, pp. 658–673, Jun. 2012.

[4] A. Nadkarni and W. Enck, "Preventing accidental data disclosure in modern operating systems," in *Proc. of SIGSAC*, 2013.

[5] S. Heuser, A. Nadkarni, W. Enck, and A.-R. Sadeghi, "ASM: A Programmable Interface for Extending Android Security," in *Proc. of USENIX Security*, 2014.

[6] M. Backes, S. Bugiel, S. Gerling, and P. von Styp-Rekowsky, "Android Security Framework: Enabling Generic and Extensible Access Control on Android," in *Proc. of ACSAC*, 2014.

[7] N. Santos, N. O. Duarte, M. B. Costa, and P. Ferreira, "A case for enforcing app-specific constraints to mobile devices by using trust leases," in *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*. Kartause Ittingen, Switzerland: USENIX Association, 2015.

[8] M. Nauman, S. Khan, and X. Zhang, "Apex: Extending Android Permission Model and Enforcement with User-defined Runtime Constraints," in *Proc. of ASIACCS*, 2010.

[9] M. Backes, S. Gerling, C. Hammer, M. Maffei, and P. von Styp-Rekowsky, "AppGuard: Enforcing User Requirements on Android Apps," in *Proc. of TACAS*, 2013.

[10] A. R. Beresford, A. Rice, N. Skehin, and R. Sohan, "MockDroid: Trading Privacy for Application Functionality on Smartphones," in *Proc. of HotMobile*, 2011.

[11] S. Smalley and R. Craig, "Security enhanced (se) android: Bringing flexible mac to android." in *NDSS*. The Internet Society, 2013.

[12] S. Bugiel, S. Heuser, and A.-R. Sadeghi, “Flexible and Fine-grained Mandatory Access Control on Android for Diverse Security and Privacy Policies,” in *Proc. of USENIX Security*, 2013.

[13] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, “TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones,” in *Proc. of OSDI*, 2010.

[14] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall, “These Aren’t the Droids You’re Looking For: Retrofitting Android to Protect Data from Imperious Applications,” in *Proc. of CCS*, 2011.

[15] Y. Zhou, X. Zhang, X. Jiang, and V. W. Freeh, “Taming Information-Stealing Smartphone Applications (on Android),” in *Proc. of TRUST*, 2011.

[16] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach, “QUIRE: Lightweight Provenance for Smart Phone Operating Systems.” in *Proc. of USENIX Security*, 2011.

[17] “Android Main Building Blocks,” http://source.android.com/devices/tech/security/index.html.

[18] “Security-Enhanced Linux,” http://selinuxproject.org. Accessed July 2015.

[19] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau, “The flask security architecture: System support for diverse security policies,” in *Proceedings of the 8th Conference on USENIX Security Symposium - Volume 8*, ser. SSYM’99. Berkeley, CA, USA: USENIX Association, 1999, pp. 11–11.

[20] “Google Play,” https://play.google.com/store. Accessed July 2015.

[21] “Softweg. Benchmark.” https://play.google.com/store/apps/details?id=softweg.hw.performance. Accessed July 2015.

[22] “QUALCOMM Trepn Profiles,” https://developer.qualcomm.com/mobile-development/increase-app-performance/trepn-profiler. Accessed July 2015.