Co-funded by the
Erasmus+ Programme
of the European Union

Mobile Device Security with ARM TrustZone

SILESHI DEMESIE YALEW

Doctoral thesis in Information Systems and Computer Engineering
Universidade de Lisboa
Instituto Superior Técnico
Lisbon, Portugal 2018
and
Doctoral thesis in Information and Communication Technology
KTH Royal Institute of Technology
School of Electrical Engineering and Computer Science
Stockholm, Sweden 2018

KTH School of Electrical Engineering
and Computer Science
SE-164 40 Kista
SWEDEN

TRITA-EECS-AVL-2018:71
ISBN 978-91-7729-962-2

Akademisk avhandling som med tillstånd av Kungl Tekniska högskolan framläggs till offentlig granskning för avläggande av doktorsexamen i Informations-och kommunikationsteknik måndagen den 14 November 2018 14:30 i Sal C Kungl Tekniska högskolan, Kistagången 16, Kista.

Tryck: Universitetsservices US AB

Abstract

Mobile devices such as smartphones are becoming the majority of computing devices due to their evolving capabilities. Currently, service providers such as financial and healthcare institutions offer services to their clients using smartphone applications (apps). Many of these apps run on Android, the most adopted mobile operating system (OS) today. Since smartphones are designed to be carried around all the time, many persons use them to store their private data. However, the popularity of Android and the open nature of its app marketplaces make it a prime target for malware. This situation puts data stored in smartphones in jeopardy, as it can be stealthily stolen or modified by malware that infects the device.

With the increasing popularity of smartphones and the increasing amount of personal data stored on these devices, mobile device security has drawn significant attention from both industry and academia. As a result, several security mechanisms and tools such as anti-malware software have been proposed for mobile OSs to improve the privacy of private data and to mitigate some of the security risks associated with mobile devices. However, these tools and mechanisms run in the device and assume that the mobile OS is trusted, i.e., that it is part of the trusted computing base (TCB). However, current malware often disables anti-malware software when it infects a device. For mobile phones this trend started more than a decade ago with malware such as the Metal Gear Trojan and Cabir.M, and continues to this day, e.g., with HijackRAT. In this work, we use the ARM TrustZone, a security extension for ARM processors that provides a hardware-assisted isolated environment, to implement security services that are protected from malware even if the mobile OS is compromised.

In this thesis, we investigate two approaches to address some of the security risks associated with Android-based devices. In the first approach, we present security services to detect intrusions in mobile devices. We design and implement services for posture assessment (which evaluates the level of trust we can have in the device), for dynamic analysis (which performs dynamic (runtime) analysis of apps using traces of Android application programming interface (API) function calls and kernel syscalls to detect apps for malware), and for authenticity detection (which provides assurance of the authenticity and integrity of apps running on mobile devices). In the second approach,

we design and implement a backup and recovery system to protect mobile devices from attacks caused by ransomware attacks, system errors, etc. Finally, we develop a software framework to facilitate the development of security services for mobile devices by combining components of the above services. As proof-of-concept, we implemented a prototype for each service and made experimental evaluations using an i.MX53 development board with an ARM processor with TrustZone.

Sammanfattning

Mobila enheter som smartphones blir huvuddelen av beräkningsenheterna på nätet på grund av deras utvecklingsmöjligheter. För närvarande erbjuder tjänsteleverantörer som finansiella och sjukvårdsorganisationer tjänster till sina kunder med hjälp av smartphonetillämpningar (appar). Många av dessa appar körs på Android, det mest använda mobila operativsystemet (OS) idag. Eftersom smartphones är designade för att transporteras runt hela tiden, används de av många personer för att lagra deras privata data. Popularitеten av Android och den öppna naturen hos dess app-marknadsplatser gör emellertid dessa till ett huvudmål för skadlig programvara. Denna situation sätter data som lagras i smartphones i fara, eftersom det kan smidigt stjälas eller modifieras av skadlig kod som infekterar enheten.

Med den ökande popularitеten hos smartphones och den ökande mängd personuppgifter som lagras på dessa enheter har säkerhet hos mobila enheter fått stor uppmärksamhet från båda industri och akademi. Som ett resultat har det föreslagits flera säkerhetsmekanismer och verktyg som anti-malware programvara för mobila operativsystem för att förbättra integriteten hos privata data och för att mildra vissa av säkerhetsriskerna i samband med mobila enheter. Men dessa verktyg och mekanismer kör på enheten själv under antagandet att det mobila operativsystemet är betrott, det vill säga att det ingår i den betrodda basen för databehandlingen (TCB). Men nuvarande malware avaktiverar ofta anti-malware-programvara när den infekterar aën enhet. För mobiltelefoner började denna trend för mer än ett decennium sedan med skadlig kod som Metal Gear Trojan och Cabir.M, och fortsätter till denna dag, t.ex. med HijackRAT. I detta arbete använder vi ARM TrustZone, en säkerhetsutvidgning för ARM-processorer som tillhandahåller en hårdvaruassisterad isolerad miljö, för att genomföra säkerhetstjänster som skyddas från skadlig kod även om det mobila operativsystemet är infekterat.

I denna avhandling undersöker vi två metoder för att adressera några av de säkerhetsrisker som associeras med Android-baserade enheter. I det första metoden presenterar vi säkerhetstjänster för att upptäcka intrång i mobila enheter. Vi utformar och genomför tjänster för hållningsutvärdering (vilken utvärderar nivån på förtroende vi kan ha för enheten), för dynamisk analys (som utför dynamisk (runtime) analys av appar med spårning av funktionsanrop och syscalls till kärnan i Android Application Programming Interface (API)

för att upptäcka appar för skadlig programvara och för autenticitetsdetektion (som ger garantier för äktheten och integriteten hos appar som körs på mobila enheter). I den andra metoden designar och implementerar vi ett backup- och återställningssystem för att skydda mobila enheter från attacker orsakade av ransomware attacker (utpressningsattacker), systemfel etc. Slutligen utvecklar vi en programvara för att underlätta utvecklingen av säkerhetstjänster för mobila enheter genom att kombinera komponenter från ovanstående tjänster. Som exempel implementerade vi en prototyp för varje tjänst och gjorde experimentella utvärderingar med hjälp av ett utvecklingskort, en i.MX53, med en ARM-processor med TrustZone.

Resumo

Os dispositivos móveis, como os smartphones, estão a tornar-se a maioria dos dispositivos computacionais. Atualmente, fornecedores de serviços como instituições financeiras e de saúde, oferecem serviços aos seus clientes usando aplicações para smartphone (apps). Muitas dessas apps são para Android, o sistema operativo (SO) móvel mais adotado atualmente. Como os smartphones destinam-se a serem carregados todo o tempo, muitas pessoas usam-os para armazenar os seus dados pessoais. No entanto, a popularidade do Android e a natureza aberta dos seus mercados de apps tornaram-o um alvo apetecível para malware. Esta situação coloca os dados armazenados em smartphones em risco, já que estes podem ser roubados ou modificados por malware que infete o dispositivo.

Com a crescente popularidade dos smartphones e o aumento da quantidade de dados pessoais armazenada nesses dispositivos, a segurança de dispositivos móveis atraiu atenção significativa da indústria e da academia. Como resultado, vários mecanismos e ferramentas de segurança, como software antimalware, foram propostos para melhorar a privacidade de dados e para mitigar alguns dos riscos de segurança associados a este tipo de dispositivos. No entanto, essas ferramentas e mecanismos são executados no dispositivo e assumem que o SO móvel é confiável, ou seja, que faz parte da base de computação confiável (TCB). No entanto, o malware atual geralmente desabilita o software antimalware quando infecta um dispositivo. Para os telemóveis, esta tendência começou há mais de uma década com malware como o Metal Gear Trojan e o Cabir.M, e continua, por exemplo, com o HijackRAT. Neste trabalho, usamos a ARM TrustZone, uma extensão de segurança para processadores ARM que fornece um ambiente isolado assistido por hardware para implementar serviços de segurança protegidos contra malware, mesmo que o SO móvel seja comprometido.

Nesta tese são investigadas duas abordagens para endereçar alguns dos riscos de segurança associados a dispositivos baseados no Android. Em relação à primeira abordagem, são apresentados serviços de segurança para detectar intrusões em dispositivos móveis. São apresentados serviços para avaliação de postura (o nível de confiança que podemos ter no dispositivo), para análise dinâmica (análise em execução de apps usando rastros de chamadas de função da API Android e syscalls) e para detecção de autenticidade (garantia da

autenticidade e integridade das apps executadas). Na segunda abordagem, é projetado um sistema de backup e recuperação para proteger dispositivos móveis de ataques de ransomware, erros de sistema etc. Por fim, é apresentada uma framework de software para facilitar o desenvolvimento de serviços de segurança para dispositivos móveis combinando componentes dos serviços acima. Como prova de conceito, foram implementados protótipos de cada serviço e realizadas avaliações experimentais usando placas de desenvolvimento i.MX53 com processador ARM com TrustZone.

Acknowledgments

This thesis could not have been completed without the support of the following people. First and foremost, my advisors Miguel Pupo Correia(IST), Gerald Q. Maguire Jr.(KTH) and Seif Haridi(KTH), for their continuous guidance, constant feedback, motivation and unconditional support throughout the course of this PhD.

I would like to warmly thank Nuno Santos from INESC-ID, for his expertise to provide much-needed guidance during the early stages of my studies when I would get stuck often.

I would also like to express sincere gratitude to Vladimir Vlassov for making my stay at KTH pleasurable and for all the support and advice he has given me throughout this work.

Finally, I am deeply thankful to my family for their love and support.

This work was supported by the European Commission through the Erasmus Mundus Doctorate Programme under Grant Agreement No. 2012-0030 (EMJD-DC) and project H2020-653884 (SafeCloud), and by national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference UID/CEC/50021/2013 (INESC-ID).

Contents

List of Figures

List of Tables

List of Acronyms

AOSP	Android Open Source Project
APK	Android Application Package
ART	Android Runtime
AXI	AMBA Advanced eXtensible Interface
BYOD	Bring Your Own Device
CFG	Control Flow Graph
CKD	Connectivity with Knowledge of Degree
CPT	Compartment Page Table
CRTM	Core Root of Trust for Measurement
DNS	Domain Name Service
DRTM	Dynamic Root of Trust for Measurement
DVM	Dalvik Virtual Machine
EPC	Enclave Page Cache
HAL	Hardware Abstraction Layer
IaaS	Infrastructure as a Service
ICC	Inter Component Communication
IOMMU	Input Output Memory Management Unit
IPC	Inter Process Communication
JNI	Java Native Interface
JVM	Java Virtual Machine
kNN	k-Nearest Neighbours
MAC	Mandatory Access Control
MMU	memory management unit
MTM	Mobile Trusted Module
NAT	Network Address Translation
NCD	Normalized Compression Distance
PC	Personal Computer

NFV	Network Function Virtualization
PCR	Platform Configuration Register
PDA	Personal Digital Assistant
ROC	Receiver Operating Characteristics
RTT	Round Trip Time
SCR	Secure Configuration Register
SDK	Software Development Kit
SEAP	Samsung Enterprise Alliance Program
SEAP	Secured Personal Device
SMC	Secure Monitor Call
SRTM	Static Root of Trust for Measurement
SVM	Support Vector Machine
TC	Trusted Computing
TCB	Trusted Computing Base
TCG	Trusted Computing Group
TEE	Trusted Execution Environment
TPM	Trusted Platform Module
TZASC	TrustZone Address Space Controller
TZPC	TrustZone Protection Controller
VLAN	Virtual Local Area Network

Chapter 1

Introduction

Mobile devices such as smartphones and tablets have gradually become an integral part of our daily lives as they enable us to access a large variety of services (e.g., checking emails, taking pictures, social networking, messaging, location-based services, gaming, online banking, entertainment, Internet browsing, video recording, e-health, etc.). The number of mobile apps has grown rapidly. Currently, many companies, from government agencies to businesses such as financial and health-care institutions, offer services to their clients using smartphone apps. Although this is convenient, it also leads people to rely on these devices to access and process privacy sensitive data (e.g., identity, passwords, financial details, health details, etc.). Unfortunately, smartphones suffer from many security issues that can put this data in jeopardy.

1.1 Motivation

Smartphones are increasingly being targeted by hackers and infected with malware. Android has become the most adopted mobile OS, with a market share much higher than all its competitors together [1]. Researchers have shown that (Android) mobile devices are vulnerable to a large number of attacks, e.g., apps and libraries that misuse their privileges [2, 3], run root exploits

that steal private information [4], take advantage of unprotected interfaces [5, 6], do confused deputy attacks [7], and do collusion attacks [8]. To improve this situation, several companies provide anti-malware software for mobile OSs. Furthermore, the research community has investigated mechanisms for detecting malware on Android using static and dynamic analysis mechanisms [9, 10, 11, 12, 13, 14]. Most of these mechanisms involve extending the mobile OS kernel or the middleware (Dalvik or ART), so their security is based on the assumption that these components are not compromised. However, such mechanisms may be ineffective as malware is often able to disable security software [15], as in the recent case of HijackRAT [16], sometimes due to vulnerabilities that allow this [17]. An alternative approach is to run the protection mechanism in an hypervisor or a thin virtual machine (VM) isolated from the VM that runs the OS and the apps, as in Droidscope [18]. However, current mobile devices do not use this kind of virtualization, as it would have a considerable overhead. This problem leads to the first research question investigated in this thesis.

Question 1: *How to design and implement security mechanisms for detecting malware on mobile devices while protecting them from malware attacks?*

The open nature of Android has led to the appearance of many alternative app markets, which facilitates the distribution of malicious apps. Some of these apps are repackaged apps, i.e., apps that were originally legitimate, but that were unpacked, modified to include a malicious payload, signed again, and then placed in a market [4]. Moreover, in some cases it has been possible to inject malicious code into legitimate Android apps without damaging the digital signature of the original app [19]. There are mechanisms in place to mitigate these problems. For example, Google Play Store analyses apps using Google Bouncer [20] and does not distribute them if it suspects they are malicious,

but some malicious apps manage to pass undetected [4, 21]. Furthermore, repackaging requires reverse engineering, which may be made harder by using code obfuscation [22], but this obfuscation is not entirely effective. Therefore, it is important to provide assurance that a mobile app is authentic, i.e., that it is indeed the app produced by a certain company and with a certain version. Authenticity is important for many apps provided by service providers, from financial to healthcare organizations, and including public administration. Some of these apps are security-critical, e.g., because they allow access to private information. The second research question investigated in this thesis is:

Question 2: *How to securely detect if an app running in a mobile device is authentic or was not modified in an unauthorized way.*

The growing popularity of Android and the increasing amount of sensitive data stored in mobile devices have led to the dissemination of Android ransomware. Ransomware is malware that prevents access to data in a computer, either by locking the system's screen or, more frequently, by encrypting files, then demanding a ransom from the victim to provide that access [23]. The ransom is typically paid in a cryptocurrency (e.g., Bitcoin or Monero).

Ransomware has begun to attack mobile devices running the Android OS similarly to what happened earlier with personal computers (PCs) [24, 25, 26]. Android ransomware can be divided in two classes: lock-screen ransomware and crypto-ransomware. *Lock-screen ransomware* (such as *Android.Lockdroid.E*) blocks user interaction with the device, for example by leveraging the SYSTEM_ALERT_WINDOW Android permission to lock the screen. An app that has access to this permission can create system-type windows and display them on top of every other app or window, making it impossible to use the device. Before Android 6.0 Marshmallow, this permission was granted automatically

to any app from Google Play Store that requested it, facilitating this form of attack [27]. Nevertheless, this class of ransomware seems to be quite specific, as it depends on particular design vulnerabilities.

The most common form of ransomware is *crypto-ransomware* (such as *Trojan-Ransom.AndroidOS.Small* or WannaCry for PCs), which encrypts files in the victim's device and demands payment to provide the decryption key. Interestingly, paying the ransom may not return the files, as sometimes the attacker does not provide the key. Moreover, files may be corrupted and it may not be possible to decrypt them. Making regular backups of the files [28] is a common solution to this type of attack.

There are several tools to perform a local or a remote backup of data and apps in mobile devices. For example, *Samsung Smart Switch* [29] and *LG Bridge* [30] transfer data (e.g., documents, videos, pictures, and contacts) and apps from a mobile device to a PC and vice-versa. Moreover, mobile OSs (such as Android and Apple's iOS) provide utilities to easily make cloud backups. However, the existing backup tools are run in the same execution environment as the malware that infects the device, hence they may be compromised or blocked by the ransomware. In fact, as mentioned above, malware is often able to disable anti-malware software and other security software. From the attacker's viewpoint, an effective approach would be for the ransomware to disable the backup tool(s) for a period of time before it starts encrypting the files, so files that have been backed up are outdated. Furthermore, in some cases ransomware is able to encrypt or delete the backup files themselves, as happened with WannaCry itself, which deletes shadow copies of a particular volume [31]. Therefore, the backups themselves have to be protected. Unfortunately, this is something that does not happen for the online solutions just discussed.

Moreover, system components, including the mobile OS kernel, may be compromised by strong attacks or damaged for several reasons. In order to recover from system failures, mobile devices, such as Android phones, are equipped with a recovery system that enables the user to perform a full recovery/restore or maintenance, such as to reset the device to factory settings or to install/upgrade the mobile OS. The recovery system allows the user to boot to a separate bootable partition (e.g., in the case of the Android platform a recovery partition) in the internal memory of the device, circumventing the normal boot process (i.e., instead of loading the typical boot image, execution is diverted to a special recovery boot image). A device enters this recovery mode by a combination of special key presses or instructions from a command line. Different mobile device manufacturers use different recovery methods, but all perform the same function: recovering a working system. However, existing systems are vulnerable to malware attacks as they are run in the same execution environment with the malware that infects the device and possibly disables the recovery system. All these facts lead to the third research question investigated in this thesis.

Question 3: *How to securely and reliably perform recovery of user data and system components after intrusions or in the event of a full system compromise?*

Recent research has began to use the ARM TrustZone security extension to provide security guarantees in mobile devises despite the mobile OS being compromised. For example, several works use TrustZone to separate small components that do security sensitive computations from the apps and the OS, by running these components in the secure world [32, 33]. Another line of research provides secure storage to protect sensitive data such as a private key [34, 35, 36]. However, most of these security mechanisms are designed and implemented from scratch, i.e., there is not a standard framework that

facilitates creating security services for mobile devices with ARM TrustZone. This leads to the fourth research question investigated in this thesis.

Question 4: *How to design and create a software framework that facilitates the development of security services that support ARM TrustZone, for improving trust on mobile devices?*

1.2 Objectives

The first goal of this thesis is to design and implement security services to increase trust on mobile devices by taking advantage of ARM TrustZone, despite the risk of the mobile OS and the mobile apps being compromised. In short, we seek to improve the security of mobile devices by implementing security services that allow detecting intrusions in mobile devices, while the services are protected from the OS, apps, and malware by leveraging TrustZone. Additionally, we aim to combine all security components from the security services in order to develop a framework that makes these components accessible to developers, with the objective of reducing the programming and testing efforts required to create security services for mobile devices.

Our second goal is to design and implement a backup and recovery system to protect mobile devices from attacks (e.g., ransomware attacks) that compromise files and/or system components including the kernel. Therefore, we aim to perform a regular backup of files to local or remote storages, thus enabling a partial or full recovery operation.

1.3 Contributions

The contributions of this thesis are as follows:

- To address Question 1, we designed and implemented a posture assessment service for Android devices. This service allows us to securely evaluate the level of trust we can have in a device (i.e., assess its posture) even if the mobile OS is compromised. For that to be possible, the posture assessment service is protected using TrustZone. This service is configurable with a set of app and kernel analysis mechanisms that enable detecting malicious apps and rootkits.

- To address Question 1, we also designed and implemented a dynamic analyser service for Android that uses traces of Android API function calls and kernel syscalls performed by an app to inspect the app for malware, and the analyser itself is protected from malware by leveraging ARM TrustZone.

- To address Question 2, we designed and implemented a software authentication service that provides assurance of the authenticity and integrity of apps running on mobile devices. The service provides such assurance, even if the mobile OS is compromised, by leveraging the ARM TrustZone hardware security extension. It uses a set of techniques (static watermarking, dynamic watermarking, and cryptographic hashes) to verify the integrity of the apps.

- To address Question 3, we designed and implemented a TrustZone based backup and recovery service for mobile devices, that is protected from malware by leveraging TrustZone and running in the secure world. It does periodic backups of files to a secure local persistent partition and pushes these backups to external storage to protect them from ransomware. Initially, it does a full backup of the device's filesystem, then it does incremental backups that save changes since the last backup. It also

performs a partial recovery when system components fail to work or a full system recovery when the device is no longer usable.

- To address Question 4, we developed a mobile device software framework that facilitates development of security services for detecting the trustworthiness of mobile devices to increase trust on mobile devices. This framework provides a set of reusable components to simplify the implementation of security services and leverages ARM TrustZone to protect the services from malware.
- As proof-of-concept, we implemented prototypes for the above services using an i.MX53 development board with an ARM processor with TrustZone, that emulates a mobile device, to do a thorough experimental evaluation of the services.

1.4 Publications

Most of the contents in this thesis are based on material previously published in the following peer reviewed conference papers.

- S. D. Yalew, G. Q. Maguire Jr., S. Haridi, and M. Correia. DroidPosture: A Trusted Posture Assessment Service for Mobile Devices In *Proceedings of the 13th IEEE International Conference on Wireless and Mobile Computing, Networking and Communications* (WiMob 2017). DOI: 10.1109/WiMOB.2017.8115762
- S. D. Yalew, G. Q. Maguire Jr., S. Haridi, and M. Correia. T2Droid: A TrustZone-based Dynamic Analyser for Android Applications. In *Proceedings of the 16th IEEE International Conference on Trust, Security and Privacy in Computing and Communications* (TrustCom 2017). DOI: 10.1109/Trustcom/BigDataSE/ICESS.2017.243

- S. D. Yalew, P. Mendonça, G. Q. Maguire Jr., S. Haridi, and M. Correia. TruApp: A TrustZone-based Authenticity Detection Service for Mobile Apps. In *Proceedings of the 13th IEEE International Conference on Wireless and Mobile Computing, Networking and Communications* (WiMob 2017). DOI: 10.1109/WiMOB.2017.8115820

- S. D. Yalew, G. Q. Maguire Jr., S. Haridi, and M. Correia. Hail to the Thief: Protecting Data from Mobile Ransomware with ransomSafeDroid. In *Proceedings of the 16th IEEE International Symposium on Network Computing and Applications* (NCA 2017). DOI: 10.1109/NCA.2017.8171377

Research works that have been conducted during my doctoral studies, but not included in this thesis are the following.

- S. D. Yalew, G. Q. Maguire Jr., and M. Correia. Light-SPD: A Platform to Prototype Secure Mobile Applications. In *Proceedings of the 1st ACM Workshop on Privacy-Aware Mobile Computing* (PAMCO 2016). DOI: 10.1145/2940343.2940349

- M. Guerra, B. Taubmann, H. P. Reiser, S. D. Yalew, and M. Correia. Introspection for ARM TrustZone with the ITZ Library. In *Proceedings of the 18th IEEE International Conference on Software Quality, Reliability, and Security* (QRS 2018).

- N. O. Duarte, S. D. Yalew, N. Santos and M. Correia. Towards Auditable Native Functions by Leveraging Verifiable Computing and ARM TrustZone. *15th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services* (MobiQuitous 2018). (Accepted)

1.5 Structure of the thesis

The rest of this thesis is organized as follows.

Chapter 2 discusses the necessary background concepts and related work required to understand the thesis contributions. This chapter provides an overview of the security problems specific to the Android platform and security mechanisms (software-based and hardware-based) implemented in mobile devices in improving the security of these devices.

Chapters 3-6 present the peer-reviewed contributions of this thesis in detail. Chapter 3 presents the design and implementation of a security service that allows to ensure trustworthiness of mobile devices. Chapter 4 presents the design and implementation of a security service that is able to analyze Android apps using dynamic analysis techniques for detecting malware. Chapter 5 presents the design and implementation of a security service that is used to verify the authenticity and the integrity of Android apps. Chapter 6 presents the design and implementation of a secure backup and recovery service for mobile devices.

Chapter 7 presents the software framework for building security services, and describes the components of the framework and their APIs.

Chapter 8 recaps the main contributions of the thesis and proposes the future work.

Chapter 2

Background and Related Work

This chapter presents background information and the related work to understand the issues and the solutions covered in this thesis. The chapter starts with the basics of the Android architecture and its security mechanisms. Next, the current security issues for Android devices and the proposed security extensions to the Android platform are discussed. Finally, trusted computing technologies with specific focus on the ARM TrustZone security extension for dealing with isolated execution, and security solutions that have been developed to take advantage of the ARM TrustZone are discussed.

2.1 Android mobile operating system

An operating system (OS) is software that is designed to manage the use of the hardware and software resources of a computer by the various programs that run on top of it. A mobile operating system is an operating system that is specifically designed to run on mobile devices such as cellular phones, personal digital assistants (PDA), and tablets. There are different mobile OSs available on the market (e.g., Java ME, Symbian, Android, iOS, Windows Phone, etc.). Today, Android and iOS are the most popular platforms. Android is an open source OS created by Google, and was designed to run on different mobile

devices [37]. In contrast, iOS is a proprietary mobile OS developed by Apple Inc. as an extension of Mac OS X, and is used only for Apple devices, such as iPhone, iPad, and Apple TV [38, 39]. In this section, we focus on Android and discuss its security architectures.

Google is not responsible for providing Android OS images for mobile devices, i.e., it produces a baseline version of Android and makes it freely available in the form of the Android Open Source Project (AOSP). The device vendors (e.g., Samsung, Apple, and Huawei) are free to build their own versions upon this baseline (e.g., adding custom features) for their devices. These customizations have grown increasingly necessary as the hardware has grown more capable.

Android is a Linux-based open source software stack for mobile devices. It consists of a modified *Linux kernel*, a *middleware layer*, and an *application layer* (as depicted in Figure 2.1).

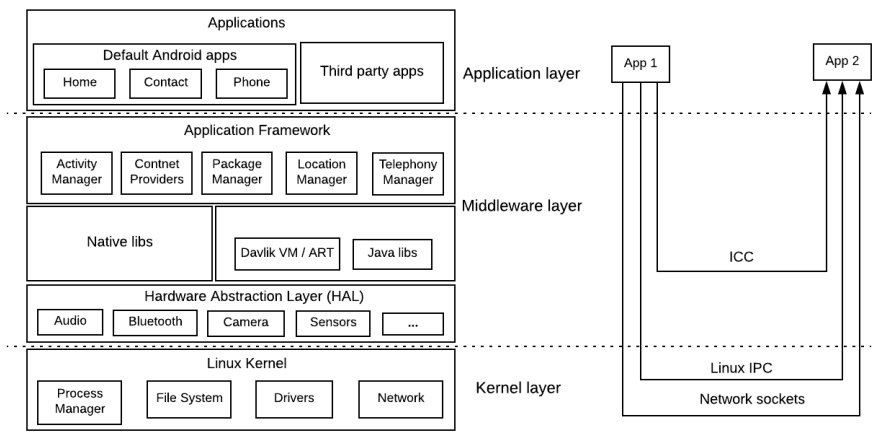

Figure 2.1: Android software stack.

The Linux kernel provides basic operating system services such as memory management, process scheduling, device drivers, file system support, and

network access. Although, Android mostly relies on the Linux kernel functionality, certain enhancements, such as Android specific drivers that are required to perform system operations, have been integrated with the Linux kernel. For example, *Ashmem* (a replacement of the standard Linux shared memory functionality that provides a means for the kernel to reclaim shared memory blocks if they are not currently in use), *Binder* (a driver that provides the support for custom Inter-Process Communication (IPC) mechanism between precesses in Android), *Wakelocks* (a mechanism that prevents the system from sleeping), *Low Memory Killer* (used to kill processes as available memory becomes low), *Logger* (Android logging mechanism that provide system wide logging facility), etc.

The middleware layer consists of the application framework, Android libraries (written in C/C++), the Android runtime environment, and the hardware abstraction layer (HAL). The application framework consists of a set of APIs for Android apps to interact with system services. Android comes with a number of system services that provide basic OS functionalities, e.g., the *ActivityManagerService* that manages the life cycle of apps and the *PackageManagerService* that manages apps installation, update, deletion, etc.

The Android runtime includes the *Android Runtime* (ART) and core Java libraries. ART is a lightweight implementation of Java Virtual Machine (JVM) optimized to run multiple virtual machines on mobile devices. Each Android app runs in its own process and with its own instance of ART. Android apps are written in Java based on the APIs the Android software development kit (SDK) provides, but they can also incorporate C/C++ native libraries through the Java Native Interface (JNI). Android apps are compiled to a custom bytecode known as the Dalvik Executable (DEX or .dex) bytecode format, which can run on the Android platform. ART compiles a .dex file into

native machine code (a system-dependent binary) once at install time, which is known as ahead-of-time compilation. This runtime environment improves the performance of the Android platform and apps. Prior to Android version 5.0, the Android runtime was based upon the *Dalvik Virtual Machine* (DVM). This earlier runtime environment is based on just-in-time compilation (JIT) which compiles a .dex file to machine code each time the app is run. If an app is developed for ART, then it should work when running with Dalvik, but the reverse may not be true.

The Android HAL provides standard interfaces/APIs for upper layers (e.g., the application framework) to interact with the underlying hardware. HAL consists of library modules, each of which implements the API defined in Android framework for a specific type of hardware component, such as the camera or bluetooth module. The hardware vendor implement HAL driver modules for their proprietary hardware. HAL allows apps and system services to interact with the kernel drivers. HAL uses system functions provided by the Linux kernel to serve a request from the upper layers to access device hardware.

The Android application layer includes core apps (i.e., apps installed by default) such as home, phone dialer, and contact provider (Contacts). The user can also install additional apps from, for example, the Google Play store. Android apps are comprised of four basic types of components (modules): *Activities*, *Services*, *Content Providers*, and *Broadcast Receivers*. Activities are responsible for the user interface. Each screen shown to a user is represented by a single Activity component. Services implement the functionality of background processes which are invisible to the user. Content Providers are special purpose components used for sharing data among apps. Broadcast Receivers receive event notifications from the system and from other apps.

Android apps are designed to work with other apps and services. Apps expose selected features/components to other apps and use those provided by others at runtime, i.e., components of one app can invoke or use components of other apps.

Android provides different communication models for apps to communicate with each other (as shown in Figure 2.1): (1) IPC based on Unix domain sockets; (2) apps with the INTERNET permission can create network sockets and; (3) a lightweight IPC mechanism between app components based on Binder driver, denoted as *Inter-Component Communication* (ICC). Moreover, app components can interact using an *Intent*, which is a message object used to request an action from another app component (e.g. start an Activity). Intent messages are delivered using the Android ICC mechanism.

Each Android app is distributed as a file in the Android app package format (APK or .apk). An APK file is essentially a zip archive containing all the app's resources: bytecodes (.dex), manifest file, media files, etc. Every app package includes a manifest file (*AndroidManifest.xml*) which provides essential information about the app, including the description of components (activities, service, broadcast receivers, and content providers) that the app is composed of. The manifest also includes information about required and granted permissions. In other worlds, it declares which permissions the app must have to access protected parts of other apps and the permissions that others are required to have to interact with the app's components. For instance, if a developer wants to provide an app with the possibility to send SMS, then the developer should add into the app's *AndroidManifest.xml* file the following line; `<uses-permission android:name="android.permission.SEND_SMS">`.

2.2 Android security mechanisms

Android incorporates security mechanisms to protect apps, data, and system resources. In this section, we discuss the core security mechanisms of Android: the *permission framework* at the middleware level and *application sandboxing* at the Linux kernel level.

2.2.1 Permission framework

To maintain the security of mobile device resources, Android's permission mechanism enforces restrictions on specific operations that an app can perform. Android requires apps to request permission before they can use certain data and features. In other words, the Android permission framework is used by the Android middleware to control access to system resources, such as the ability to access the camera or the network, and app components, such as the ability to invoke a service provided by another app. The permissions/security policies are statically defined by Android app developers in the apps' manifest file, i.e., the developer has to specify the set of permissions required by the app and permissions used to restrict access to each app components.

Android contains a set of predefined permissions for the system services that control operations ranging from dialing the phone (`CALL_PHONE`), reading SMS (`READ_SMS`), using the Internet (`INTERNET`), and listening to key strokes (`READ_INPUT_STATE`). In addition, any Android app can also define new, custom permissions to protect its own interfaces/components (such as activities and content providers).

To obtain a permission, Android requires apps to explicitly request it. Depending on how sensitive the resource is, the system may grant the permission

automatically, or it may ask the user to approve the request. A permission can be associated with one of the following four permission/protection labels:

Normal: These permissions are granted by the system without asking for any permission from the user.

Dangerous: These permissions ask for approval from the user. Before Android version 6.0, a user has two choices, either grant all permissions requested or deny all at the time of installation. Denying permissions will stop the installation of the app.

Signature: The system grants this permission if the requesting app is signed with the same certificate as the granting app.

Signature system: The same as the signature, but only used for system apps. This permission is granted only if its requesting app is in the same Android system image, which includes the code for the Android Linux kernel, the native libraries, the Dalvik VM, and the various Android packages (such as the Android framework and preinstalled apps).

In Android below version 6.0, users have to either grant all permissions an app requests at install time or abort the installation process. Permissions cannot be selectively granted or denied. Moreover, if the user has approved permissions, the user cannot revoke them. The only way to remove the permission is to uninstall the app. However, beginning in Android 6.0, users grant permissions to apps during runtime, they do not need to grant permissions when they install or update the app. This gives the user more control over the app's functionality; for example, a user could choose to give an app access to the camera but not to the device's location. The user can also revoke the permissions at any time, so even if the app used the camera yesterday, it

cannot assume it still has that permission today. If an app tries to perform an operation for which it has no permission (e.g., send SMS), Android will typically throw a security exception back to the app.

2.2.2 Application sandboxing

By taking advantage of the security features offered by the Linux kernel, Android uses *discretionary access control* to isolate apps from one another in much the same way discretionary access control is employed in conventional Linux systems to isolate users on a multi-user system. To this end, each app is assigned a unique user (UID) and group (GID) identifier when it is installed. When an app creates a file, file permissions are by default set to `'rw-rw---'` (`0660` in octal notation). This means that only the app that owns the file or apps with the same GID of the owner/creator are able to read and write the file, other apps installed with different UID and GID cannot read, write, and execute the file.

Access to low-level system resources, such as device drivers and network sockets are also enforced by the Linux kernel by checking the UID or GID of the process for an app against the resource's owner and access mode. For example, the permissions to a device camera (`/dev/cam` device driver) are set to `'rw-rw---'`. This means that only processes run as root or which are included in camera group are able to read from and write to this device driver. Thus only apps, which are included into camera group, can interact with the camera.

The mappings between permission and GID for built-in permissions are defined in the `/etc/permission/platform.xml` file. The package manager reads `platform.xml` at boot and maintains a list of permissions and associated GIDs. When it grants permissions to a package, it checks whether each

permission has an associated GID. If so, the GID is added to the list of GIDs associated with the app in `/data/system/packages.list` file. Thus, if an app has requested the access to a camera feature and the user has approved it, this app is also assigned a camera Linux group GID, therefore this app will be able to interact with the camera's device driver.

Although the Linux kernel sandboxes apps and prevents them from accessing other apps' data, they can share their resources (e.g., activities, content providers, or other components) by declaring permissions in their manifest file. If an app wants to share its data or files with other apps, the app needs to develop its own content provider. Furthermore, apps can share their resources without restriction and even run in the same process by sharing the same UID, called a shared user ID. To share the same UID, apps need to be signed by the same key and explicitly specify in their manifest file the `sharedUserID` attribute. Moreover, apps can use the `MODE_WORLD_READABLE` and `MODE_WORLD_WRITEABLE` flags when creating files to make them publicly accessible (e.g., by other apps with different UIDs and GIDs).

2.3 Security issues

Improvements in hardware and software have enabled ever more complex tasks to be performed on mobile devices, together with the fact that these devices often store sensitive personal data, has exposed them to a high level of risk. Whenever a new feature of mobile devices emerges, it generally introduces new opportunities for new threats. Attacks on a mobile device can make the phone partially or fully unusable, allow a malicious user to remotely control the device, or allow another party to steal private information stored on the device.

A major source of security problems in Android is malware. The increasing popularity of Android and the open nature of its app marketplaces facilitate the distribution of malware [2, 3]. Malware is mainly distributed either in the form of malicious apps or repackaged apps [4].

The majority of malicious code distributed for Android is disseminated through third-party app stores [40]. Android users can easily opt to download and install apps from third-party marketplaces, and no security scan is performed by Android when installing apps, rather it provides the user with the power to decide whether an app should be trusted or not. As a result, the user will undoubtedly install suspicious apps or allow app inappropriate permissions if they are not paying attention. Some apps, even when downloaded from the official Android market, the Play Store, can be malicious [4, 21].

Malware can be also distributed through other attack vectors, such as an email with a malicious attachment, a link in an infected website, an SMS containing a link to a website where a user can be tricked to download the malicious code, an MMS with infected attachments, or infected programs received via Bluetooth.

In this section, we discuss the main types of threats and vulnerabilities to Android devices to understand the security problems affecting these devices.

2.3.1 Overclaim of permissions

Since users are unaware of all the functionalities of apps, apps may request and be granted more permissions than they require. This leads to spyware attacks on mobile devices that expose users to potential private data leakage. Spyware is a malicious app installed with the user's consent, which requests and abuses more permissions than those required for the stated purpose of the app. For instance, a standalone game app may request the `SEND_SMS` permission which

can be exploited to send premium SMS without the user's consent. Moreover, advertisement libraries, often included in third-party apps, have been shown to exploit the permissions of their host app to collect information about the user, including privacy sensitive data such as the user's contacts or location [2, 3, 41].

2.3.2 Permission escalation attacks

Although the sandboxing mechanism provides apps isolation based on a private UID, at runtime a malicious app can escalate its permissions by collaborating with other apps to access critical resources without explicitly requesting the requisite permission [7, 42, 8, 43]. Permission escalation attacks can be classified into two categories: confused deputy and collusion.

Confused deputy attack. Such an attack leverages unprotected interfaces of privileged poorly designed benign apps to indirectly access protected functionality or data and thus effectively escalate an app's permissions at runtime [44].

Collusion attacks. Another source of vulnerability arises from the shared UID feature. If the *sharedUserID* attribute in the app manifest file is set to the same value for two or more apps developed by the same developer with the same certificate, they will share the same UID. Each of these apps can access the other's data and will have all their granted permissions, i.e., the combined set of permissions. For instance, a user has installed two apps sharing a common UID and the first app has permission to access the Internet and the other one can access the user's contact list, then each app is capable of both reading the user's contacts and sending them through the Internet. However, the user will be unaware of this collaboration between these apps.

2.3.3 Root exploits

Several vulnerabilities have been being found in the kernel layer of the Android framework [45, 4]. Exploiting vulnerabilities grants the attacker root privileges on the system. Attackers can use root exploits to gain extra privileges and perform any operation on the phone. For example, an attacker can use a root exploit on an Android phone to gain access to privacy sensitive data that should have been protected by the permission system, e.g., accessing the contact list directly from the contacts' provider.

2.4 Software-based security mechanisms

To improve the security of Android, a number of software-based security mechanisms have been proposed. These are described in this subsection.

2.4.1 Static and dynamic analysis

In the default Android permission framework, permissions for apps to access privacy sensitive information are granted by mobile users. However, users lack visibility into how apps use their private data. To address this issue, there has been much research on methods for analyzing the malicious behavior of Android apps. These methods can be roughly categorized in two approaches: *static* and *dynamic* analysis.

Static analysis methods detect if an app is malicious by inspecting its code (e.g., extracting class names, snippets of code, checksums, functions names, etc.) or metadata (e.g., information extracted from `AndroidManifest.xml` file, such as required Android permissions), without executing the app [9, 10]. The extracted information is then compared with sets of known clean and malicious apps to determine if they have any similarity to the new app. For

instance, FlowDroid [46] and ComDroid [47] detect privacy sensitive data leaks by scanning the source code or bytecodes. This approach allows examining all possible execution paths in the program, not only those invoked during execution, i.e., all the Android app's code is analyzed even code that will never or may never be executed at runtime.

Kirin [9] extracts requested permissions of an app from the app's manifest file and checks them against a set of security rules. Similarly, Stowaway [10] analyses Android function API calls to detect over-privileged apps. Signature-based systems, such as DroidAnalytics [48] and Androguard [49], have a database of distinctive patterns (signatures) of malicious code and look for them in apps to detect malicious apps or injected malicious code. Other mechanisms, such as Drebin [50], analyze selected features (e.g., requested permissions and sensitive function calls) extracted from sample malware and benign apps, using machine learning classifiers to detect whether an app is malicious or not. However, the static analysis mechanisms can be evaded by exploiting dynamic code update features (e.g., using `DexClassLoader`), i.e., an Android app can load, additional code from external sources at runtime and there is no security check of this external code. Furthermore, the dynamically loaded code runs with the same security permissions as the host app. This gives malware an opportunity to add malicious code into the device. However, an app can be statically analyzed to identify whether the app can load additional code during runtime [51].

Dynamic analysis methods, on the contrary, detect malicious apps by evaluating their behavior during execution [52, 53, 54, 11, 12, 13, 14]. Some works control the information flow in real-time to prevent the exposure of privacy sensitive data [52, 53]. For example, TaintDroid [52] tracks the flow of sensitive data and generates an alert if such data is passed to the network stack. Such

real time feedback gives users and security services the ability to analyse apps' behavior in order to identify malicious apps.

TaintDroid [52] assumes that downloaded third party apps are not trusted, hence it monitors how these apps access and manipulate the users' personal data, such as GPS location information and address book information. TaintDroid automatically labels (taints) data from privacy sensitive sources (taint source) and then transitively applies labels as sensitive data propagates through program variables, files, and inter-process messages. If any tainted data leaves the system via a network interface (taint sink), TaintDroid raises an alert and provides log information about the data's labels, the app responsible for transmitting the data, and the data's destination.

AppFence [55] and AppGuard [56] also try to mitigate privacy leaks by dynamically monitoring apps. AppFence is built upon TaintDroid [52] and can block unwanted data transmission. It implements two runtime mechanisms to protect users' privacy: data shadowing and blocking sensitive data from being exfiltrated off the device by extending TaintDroid. Similar to MockDroid [57], AppFence provides blank or fake data to an app when the app attempts to access private data such as contact list and location information. It also controls network transmission of sensitive data and blocks this transfer if the data is only available to the app for on-device use by intercepting calls to the network stacks to detect when such data is written to a socket.

The above approaches could negatively affect the functionalities of mobile devices. Apps that violate the privacy requirements defined by these approaches may indeed provide user-desired functionalities. For instance, transmission of sensitive data does not always indicate privacy leakage. A mobile user may intentionally send out private data, such as location and contact information to remote cloud services using legitimate apps. In order to provide an improved

solution to this problem, AppIntent [53] presented a privacy leakage detection mechanism for Android apps that allows it to distinguish whether a privacy leak is user-intended or not. For sensitive data transmission from a privacy sensitive source, AppIntent provides context information in the form of the GUI and the description of the user's action, which helps to determine if the data transmission is intended by the user or not. This context information is derived from input data and user-triggered events.

2.4.2 Fine-grained permissions

A collection of works [9, 58, 59, 60] extends the current Android permission framework to address the issues raised due to Android's coarse-grained permissions. A finer-grained access control system could be used to ensure that any app is granted the least privilege necessary for its correct operation.

Finer-grained access control permissions can be enforced both at install time and runtime. At install time, a finer-grained policy enforcement may be used to stop malicious apps from being installed on a user's device. Different factors such as permission combinations and signatures of requesting apps can be taken into consideration to define security policies, and the installation process is aborted if an app violates one of these predefined policies [9, 58]. Runtime policy enforcement provides finer-grained restrictions at runtime [59]. This type of finer-grained restriction can be implemented by modifying the permission configuration of an installed app, such as how many times an app with `SEND_SMS` permission is allowed to send an SMS message within a given period of time, and what kind of private information can be accessed.

Kirin [9] is an extension to Android's app installer, which exerts fine-grained control over the assignment of permissions at install time using a set of predefined security rules to identify dangerous combinations of permissions

requested by apps. Kirin extracts the list of the permission configurations and action strings from an app's manifest file at install time, and checks whether these permissions violate its predefined security rules. For example, an eavesdropper to be notified of voice call activity may require a combination of `READ_PHONE_STATE`, `RECORD_AUDIO`, and `INTERNET` permission labels. The security rules are manually defined to detect undesired permission combinations, which might be dangerous and misused by malicious apps. However, apps may provide unprotected interfaces that can be accessed by other apps without requiring any permission at install time. Hence, runtime policy enforcement should be imposed to address this issue.

Similarly to Kirin [9], Secure Application INTeraction (Saint) [58] controls the granting of app defined permissions using a set of security policies. In addition to Android's permission system, an app can enforce security decisions based on signatures, configurations, and contexts (e.g., phone state) both at install time and at runtime, i.e., an app developer can add security features to their apps that will be enforced to regulate install time Android permission assignment and their runtime use. This gives app developers the possibility to protect app's interfaces, from being misused by malicious apps. A signature-based policy could be required to control what permissions are granted based on the signature of a requesting app, and also a configuration-based policy could be required to control the permission assignments based on the configuration parameters of a requesting app, such as app versions. At install time, the permission is retrieved from the manifest file in the requesting app APK by a Saint-enhanced Android installer. For each permission, this installer queries an AppPolicy provider, which maintains a database of all the install-time and runtime policies. The AppPolicy provider consults its policy database and returns a decision according to its matching rules. If the policy conditions hold, then the installation proceeds. Otherwise, it is aborted. Upon successful

installation, the new app's policies are appended to the AppPolicy provider's policy database.

In addition to install time policy enforcement, Saint also enforces runtime policies which regulate ICC calls based on a configuration-based policy, and phone context-based policy. For example, the phone context based policy is used to control communication between apps based on phone contexts, such as location, time, Bluetooth connection and connected devices, call state, data connection network, and battery level. When a caller app initiates an ICC call, the ICC call is intercepted by the policy enforcement code before any Android permission evaluation. The ICC call is allowed to continue if all security policies provided by the caller app and callee app are satisfied.

Apex [59] proposes an extension to the Android permission framework allowing users to selectively grant permissions requested by an app at installation time, and also specify user defined constraints for each permission, e.g., when and how many text messages can be sent per day. With Apex, even after an app has been installed, the user is able to grant additional permissions or revoke some of the granted permissions. In Apex, the source of security policies is mainly the user. In other words, the user is responsible for granting or denying permissions. Therefore, Apex relies on the user to make appropriate security decisions. Similarly to Apex, Flex-P [61] gives the user the possibility to grant a subset of permissions rather than asking the user to grant all permissions or nothing. The user can also change the granted permissions at any time, even after installation.

CRePE [60] presents a solution that attempts to restrict an app's permissions depending on the context of the phone such as location, time, and temperature. For example, employers could enforce a company-wide policy for all employees to restrict the set of apps that could run on smartphones provided by the

company to its employees only during working hours and while located at the company's location(s).

2.4.3 ICC call tracking

Android's security extensions such as Kirin, Saint, TaintDroid, and CRePE, do not address privilege escalation attacks in general, since they mainly focus on enabling or disabling of certain functionalities rather than on transitive permission usage across multiple apps.

QUIRE [44] provides a lightweight provenance system that prevents confused deputy attacks. At runtime, when the Android reference monitor intercepts an ICC call, QUIRE tracks and records the ICC call chain associated with the request and denies access if the originating app has not been explicitly granted the corresponding permission. Similarly to QUIRE, IPCInspection [42] keeps track of information flows through ICC to reduce the permissions of an app when it receives a message from another app with less privilege. The permissions of the app are reduced to the intersection between the recipient and the requester permissions. However, neither QUIRE nor IPCInspection address privilege escalation when colluding apps are malicious. Maliciously colluding apps may forge the ICC call chain to obscure the originating app.

Similarly to QUIRE and IPCInspection, XManDroid [62] inspects information flows over ICC calls, and makes policy decisions based on security rules defined in a system policy database. XManDroid extends the Android reference monitor and verifies whether the requested ICC call complies with the predefined security policies when the default Android reference monitor grants the ICC call. In addition to the permission based approach, XManDroid makes policy decisions based on the content of Intents (messages that are passed via ICC

calls). Moreover, XManDroid may request user confirmations in order to allow or deny a particular ICC call.

2.4.4 Mandatory access control

The above solutions extend Android, but only attempt to address access control at the Android middleware layer; thus they do not solve potential weakness at the kernel level or the limitations of its DAC mechanism.

TrustDroid [63] and XManDroid [64] adapted Tomoyo Linux [65], which is a mandatory access control (MAC) framework for Linux to enforce a kernel-level MAC on the file system, IPC, and network sockets.

TrustDroid [63] provides domain isolation on Android, typically between a business domain and a private domain (i.e., trusted and untrusted domain) to protect an enterprise's (a company's) assets. In an enterprise that provides mobile devices to its employees to access the company's apps and data, the company data could be compromised if an employee installs malicious apps on thier assigned device. Therefore, corporate assets should be isolated in a separate domain from untrusted apps. At installation time, TrustDroid [63] colors an app (i.e., labeling an app as trusted or untrusted) by checking a certificate provided by the company and contained in the app's APK. Based on the apps' colors, TrustDroid classifies apps along with their data into logical domains. TrustDroid enforces isolation on many abstraction layers of the Android software stack: (1) in the middleware to prevent any data exchange or an ICC call between different domains (similarly to MOSES [66]), (2) at the kernel layer by enforcing MAC on the file system and on IPC traffic, and (3) at the network layer to regulate network traffic, e.g., denying Internet access to untrusted apps while the employee is connected to the company's network.

SEAndroid [67] and FlaskDroid [68] leverage SELinux [69] to implement MAC on the kernel level. SELinux is a Linux kernel security module, which supports a type enforcement policy language to control access to resources and services by enforcing MAC over all processes including process running with root/superuser privileges, thus mitigating the potential damage from malicious apps. SEAndroid integrates the SELinux implementation in the Android's kernel and places all the processes, files, sockets and other kernel resources under the control of the MAC. Thus, SEAndroid limits root and other users' privilege.

FlaskDroid extends type enforcement to Android's middleware layer on top of SEAndroid. FlaskDroid [68] provides fine-grained access controls to objects (e.g., kernel resources such as files or IPC and middleware resources such as services, intents, or content provider data). On access to these objects by subjects (i.e., apps) to perform a particular operation, it enforces an access control decision via request to the SELinux security server in the kernel.

2.4.5 Component-level access control

Permissions are granted to apps regardless of what each component contained in an app needs. Each component shares the same set of permissions with the app, so some components in the app might have more privilege that they need. Apps often include third-party components, such as advertising libraries (e.g., Google's AdMob). This may be risky as third-party components have been known to abuse app's permissions to collect privacy-sensitive information without user consent [4].

To address this problem, several works [70, 71, 72] propose assigning separate permissions to the in-app advertisement component to minimize the privileges available to advertising libraries.

AdDroid [70] separates advertising components by encapsulating them as a service and creates a new advertising API and corresponding advertising permission. This allows apps to show advertisements without requesting privacy-sensitive permissions.

AdSplit [72] provides process-level isolation of the advertising component from the host app. It allows the advertising component to run entirely in a separate process with limited permissions. Thus the host app no longer needs to request permissions on behalf of their advertisement libraries.

Compac [73] further generalizes the in-app privilege separation solution by proposing a component-level permission assignment approach, in which each in-app third-party component only gets the minimum set of permissions needed for preserving the app's functionalities. However, this approach is not applicable to third-party libraries relying on JNI or dynamic code execution, which almost all the popular third-party libraries rely on. FlexDroid [74] provides in-app privilege separation and provides app developers with fine-grained access control for third-party libraries. It relies on call stack traces collected at both Dalvik VM and native levels and is suitable for sophisticated libraries.

2.4.6 Inlined reference monitors

Two works [75, 76] place security and privacy policy enforcement code (i.e., a reference monitor) inside apps to achieve access control through code instrumentation or rewriting, instead of relying on system-level hooks, so that there is no need to modify Android to enhance its security and privacy controls. For example, Aurasium attaches sandboxing and policy enforcement codes to the app package (which may have been downloaded from an unknown source) in order to dynamically monitor the app's behavior for security and privacy

violations. Aurasium monitors almost all types of interactions between the app and the operating system. For instance, Aurasium performs security and privacy policy evaluation when an app attempts to access an SMS message, contact information, or services such as camera or GPS. However, in contrast to a system-centric solution, inlined reference monitoring has a drawback. The reference monitor and the potentially malicious code share the same sandbox, hence the monitor is no more privileged than the malicious code and thus prone to being compromised.

2.5 Trusted computing

The notions of trusted computing (TC) appeared with technologies developed and promoted by the *Trusted Computing Group* (TCG) [77] to improve computer security through hardware mechanisms and the associated software.

2.5.1 TCG work

TCG developed a specification for a hardware security component called the *Trusted Platform Module* (TPM) [78] for PCs. A TPM is a hardware chip designed to provide some security services, such as cryptographic operations (e.g., hashing and signing) and secure storage of critical data (e.g., called measurements) using a set of Platform Configuration Registers (PCRs). The TPM chip is usually installed/mounted on the motherboard of commodity PCs and communicates with the rest of the system, including the main CPU, using a hardware bus. TPM allows a *measured boot*, i.e., a boot method where each component is measured by its predecessor before being executed, to verify the integrity of OS and user apps. TPM implements *Static Root of Trust for Measurement* (SRTM), which takes place at system boot. The boot of a system involves executing a sequence of components (BIOS, Bootloader,

kernel, etc). The first component executed (e.g., the BIOS in ROM) is called the *Core Root of Trust for Measurement* (CRTM), as the integrity of the whole process depends on the trust we can have in that component [79]. The CRTM is implicitly trusted and measures the next component in the boot sequence and stores its measurement (a cryptographic hash value) into the TPM at PCR 0. This process is executed for each component in the boot sequence. This measurement process enables TPM-based remote attestation and data sealing.

TCG has also developed a specification for a *Mobile Trusted Module* (*MTM*) [80], which is a version of the TPM to increase the security of mobile devices. The MTM specification is platform independent, allowing a wide range of potential implementations for manufactures. MTM-based security services can be offered in dedicated hardware, software, or a hybrid solution. The MTM provides a root of trust for mobile devices in the same way as the TPM does for PCs.

The authors of [81] proposed a remote attestation mechanism using MTM to verify the integrity of an OS's kernel and to ensure that no malicious apps are running on an Android-based device. For this purpose, they created an MTM emulator in software, as at the time there was no hardware implementation of MTM for mobile devices. The basic idea is to measure all the executables on top of the virtual machine as well as on the kernel level and then report these measurements to the remote attestation service. For verification purposes, the hash values of the executables of apps from an Android market are stored in a database and classified as benign apps or malicious apps. Later, when verifying the integrity of the system, the remote attestation system compares the hash values reported by the MTM with the hash values in the database.

Another line of work on TC aims at ensuring that sensitive code and data are stored, processed, and protected in an isolated environment called a *Trusted Execution Environment* (TEE). A TEE runs alongside but is isolated from the device's main OS (e.g., Android). Researchers have realized TEEs based on virtualization, e.g., Terra [82], that rely on a hypervisor to create and manage TEEs without any modification to existing hardware.

2.5.2 ARM TrustZone

More recently hardware extensions appeared that support the creation of TEEs. One example of a hardware-based TEE is ARM TrustZone [83, 84], a set of hardware security extensions to ARM CPUs. Unlike the TPM, this extension is part of the CPU itself, not an external chip. The ARM TrustZone extension partitions all of the System-on-Chip (SOC) hardware and software resources into two execution environments and provides isolated execution of apps and secure storage. In contrast to TPMs, which were designed as fixed-function devices with a well-defined set of services, TrustZone is used to implement a wide range of security services by leveraging the CPU as a freely programmable trusted platform. In this section, we will present the ARM TrustZone in some details as it is the main technology we used in this work. Moreover, an MTM can be implemented on top of it.

ARM is a company that creates CPU designs. However, it does not produce CPUs, but rather licenses the designs to companies that produce them. Starting with ARMv6, these designs include the TrustZone extension. This extension provides two trust domains, or worlds: the *secure world* and the *normal world*. System designers can leverage TrustZone to run a smaller kernel and some security services in the secure world, while running untrusted software in the normal world. The normal world usually runs a common OS

(e.g., Android) and its apps. The secure world should not run arbitrary code (such as an app downloaded from the Internet), but rather runs only security services. TrustZone technology is available on most modern smartphones (e.g., Galaxy S3/S4/S5/S6, Nexus 4/5, Galaxy Nexus, GalaxyNote4, and MotoG). The architecture of a TrustZone-enabled mobile device is shown in Figure 2.2.

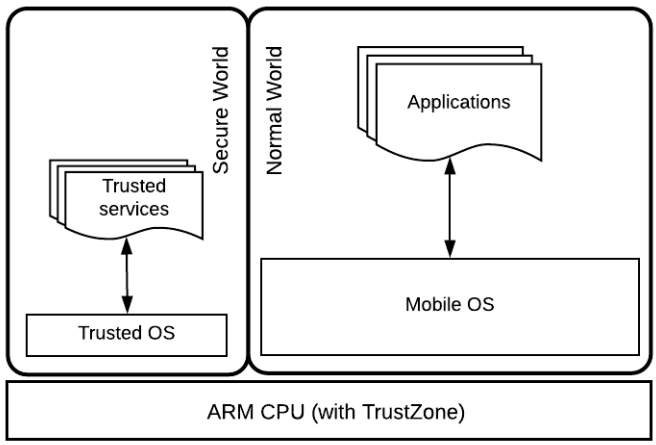

Figure 2.2: ARM TrustZone architecture.

There is an extra 33rd bit in a TrustZone enabled processor called the *non-secure bit* (NS bit) to signal in which world the processor is currently executing (i.e., the currently active world). If the NS bit is set to 1, the CPU is in the normal world. This NS bit is propagated through the *AMBA Advanced eXtensible Interface* (AXI) system bus to all memory system transactions, including access to system memory and peripherals, so the processor can enforce permissions on access to ensure that no secure world resources can be accessed by the normal world components.

The processor only runs in one world at a time, hence to run in the other world requires a context switch. In order to control the context switch between

the two worlds, TrustZone provides the *monitor mode* which preserves the processor state during the world switch. If the normal world requests to switch to the secure world, the CPU must first enter monitor mode. The monitor mode ensures the state of the world that the processor is leaving is safely saved, and the state of the world the processor is switching to is correctly restored. The processor switches to the secure world when certain exceptions occur. Software in the normal world can force a switch to the secure world either by an explicit call via a dedicated assembly instruction called *Secure Monitor Call* (SMC) or interrupts, e.g., normal interrupts (IRQ) and fast interrupts (FIQ) configured to be handled in the secure world. To return back to the normal world from the secure world, the software in the secure world can perform a return from an exception and set the NS bit in the *Secure Configuration Register* (SCR), which is a read/write register that is only accessible by the secure world, i.e., access to SCR from the normal world results in an exception.

Apart from the extensions to the processor core, hardware designers can utilize several TrustZone-based components (such as the *TrustZone Protection Controller* (TZPC) and *TrustZone Address Space Controller* (TZASC)) to logically separate the system's resources between the two worlds. TZPC can be used to configure peripherals to be accessed only by the secure world. For example, to build a secure IO path, a designer can configure TZPC to assign peripherals such as a keypad and display to the secure world. Memory space can be also split into two worlds by configuring the TZASC, so when the processor is in the normal world, it can only see the physical memory of its own world. However, the secure world can access all the physical memory available in the system. Cache memories are TrustZone-aware, i.e., cache lines can be tagged as secure or non-secure, access to secure cached content from the normal world is always denied. Furthermore, an ARM processor also provides *memory management unit* (MMU) to perform the translation

of virtual memory addresses to physical addresses. To maintain separated page translation tables between the two worlds, each world has access to its own MMU. In general, the secure world provides code and data integrity and confidentiality because untrusted code running in the normal world cannot access the resources of the secure world.

2.5.3 Other TEEs

Similar counterparts of ARM TrustZone in the PC world are the Intel Software Guard Extensions (SGX) [85, 86, 87] and its predecessor Intel Trusted Execution Technology (Intel TXT, formerly known as LaGrande Technology) [88].

Intel TXT [88] extends the processor with a set of new instructions (such as GETSEC and SENTER) for placing the measurement of a piece of critical code into the TPM and executes the code in an isolated environment. This enables a *Dynamic Root of Trust for Measurement* (DRTM), also known as late launch. Unlike TPM-based root of trust, Intel TXT allows the dynamic PCRs (PCRs 17-23) to be reset at any time while the system is running. In contrast, in TPM-based root of trust, PCRs can only be reset by a full reboot of the machine.

Similarly to Intel TXT, SGX defines a new set of CPU instructions to enable the execution of security sensitive application logic in a protected *enclave*, an isolated region of code and data within an application's address space. The access to an enclave's memory by untrusted code including the malicious OS and hypervisor is prohibited by the CPU, only code executing within the enclave can access data within the same enclave. However, enclaves can access their hosting application's memory for data exchange. With SGX, an application can create an enclave by executing a sequence of instructions, such

as ECREATE to allocate a region of virtual memory within the application for hosting the secure code and data, and EADD to add security critical code and data pages to the enclave. Similarly to its predecessors, TPM and Intel TXT, SGX supports remote attestation that enables enclaves to prove that a claimed enclave is indeed running inside trusted SGX hardware. The attestation produced by TPM covers all the software components (e.g., OS and user applications) running on a PC. In contrast, the amount of code covered by the Intel TXT and SGX attestation cover the code running in the isolated environment and the enclave code, respectively. Unlike ARM TrustZone, SGX can have multiple protected enclaves in a system, their memory areas are transparently encrypted, and an enclave is run from within a regular process's runtime context. Some important differences between SGX and TrustZone are: the first supports several TEEs and the second only one per device; the first has a much more complex interface than the second (many functions versus just one); the TEEs of the second can get exclusive access to arbitrary hardware resources whereas those of the first are more limited.

Similarly Iso-X [89], an academic work, is not available in current processors. Iso-X aims to provide protection for security-critical fragments of an application and associated data against malicious system software by using Compartment Page Tables (CPTs) (page mappings) to create and dynamically map memory pages for the compartments (enclaves). Iso-X offers higher memory allocation flexibility than SGX. In the first version of SGX (SGXv1), all enclaves are allocated in a reserved memory region in DRAM called the Enclave Page Cache (EPC), which is a fixed size dedicated memory area and the size is configured at system boot, cannot be adjusted during execution. A memory access to the EPC is controlled by the processor. Similar to SGXv1, the second version of SGX (SGX2) uses a fixed EPC size. The relevant difference between SGXv2 and SGXv1 is the ability to dynamically allocate memory

pages to an enclave at runtime. In contrast, Iso-X allows compartment pages to be located anywhere in memory and controls access to them via CPTs.

2.6 TrustZone-based security mechanisms

Recent research has begun to use TrustZone to provide security guarantees in mobile devices despite the mobile OS being compromised. TrustZone, which provides the TEE, has been adopted to secure a number of apps, of which we highlight six major areas: secure computation of sensitive app components, secure storage of critical data, secure mobile payments, system integrity verification, real-time kernel protection, and enterprise mobile security solutions.

2.6.1 Security sensitive computations

Several research works have leveraged the ARM TrustZone security extension to enable secure computations in mobile devices by splitting the app's functionalities between the normal world and the secure world [32, 33].

The Trusted Language Runtime (TLR) [33] provides a framework to separate app security logic, called a trustlet, from the rest of the app, by running it in the secure world. The trustlet is never exposed to apps and the OS in the normal world. TLR implements a small runtime capable of interpreting .NET managed code inside the secure world. A secure app can package the code handling sensitive data into a software component and run it in a secure environment, called a trustbox in the secure world.

In order to improve the security issues in the cloud computing environment, TrApps [90] uses TrustZone to implement a platform that enables trusted execution of a small sensitive component of an app that is tailored to run on cloud servers.

2.6.2 Secure storage

Recent research on secure storage of sensitive data on mobile devices has begun to utilize the ARM TrustZone extension in a number of ways. The authors of [34, 35, 36] have proposed mechanisms to protect secrets such as private keys, which are only accessible to small security critical programs in the secure world.

DroidVault [91] implements a secure storage platform for apps that stores and manipulates security sensitive data on Android devices protecting it even from a compromised kernel using TrustZone. DroidVault encrypts personal sensitive data downloaded from a remote server and stores the ciphertext in the untrusted Android filesystem, while manipulating the unencrypted data only occurs in the secure world.

TrustOTP [92] also leverages TrustZone to protect the confidentiality of one-time passwords or tokens against a malicious mobile OS.

2.6.3 Secure mobile payments

In the context of a secure payment service, the authors of [93] have proposed a location-based second-factor authentication for mobile payments by using TrustZone to verify the location of a cardholder during payments at point of sale using the phone's location. The location of the cardholder is obtained by reading the GPS coordinates from a trusted GPS sensor protected by TrustZone, in the cardholder's smartphone and a card issuer can authorize or deny the transaction based on the location of the cardholder. The card issuer matches the location of the cardholder's smartphone with the location of the terminal used for the transaction.

Light-SPD [94] and TrustPAY [95] also leverage TrustZone to enable secure payments without using physical credit/debit cards and realize privacy friendly payment. Additionally, Samsung Pay [96], a real payment service, uses ARM TrustZone to protect transaction information from attacks.

2.6.4 System integrity verification

Several mechanisms use TrustZone to associate measurements of the normal world with specific data. Measurements are hashes of the software running in the normal world obtained using a collision-resistant hash function. For example, some mechanisms provide login data [97] and sensor readings [98] together with measurements of the normal world. Software in the normal world obtains the login or sensor data, then calls the secure world to get measurements and signatures. The recipient of this information can check if the normal world is in a trusted state by checking if the values of the hashes correspond to trusted configurations. This is a direct extension of remote attestation mechanisms that have been proposed earlier for the TPM.

2.6.5 Real-time kernel protection

TZ-RKP [99] provides real-time protection of a mobile OS kernel running in the normal world, using the ARM TrustZone secure world. TZ-RKP replaces certain system functions (e.g. system control instructions, updates to the memory translation tables) in the normal world OS with hooks, thereby execution of these functions is forced to take place in the secure world. TZ-RKP also uses a memory protection technique based on a non-bypassable event-driven monitoring approach to prevent attacks that aim to modify the OS kernel running in the normal world.

2.6.6 Enterprise mobile security solutions

Currently, ARM TrustZone technology is being utilized by enterprises to develop secure platforms, such as Samsung KNOX [100], and Apple Secure Enclave [101] to enhance the security of mobile devices.

Samsung introduced a mobile security platform, called KNOX that provides a secure execution environment alongside the user's personal environment by using ARM TrustZone. KNOX allows running enterprise apps and data in an isolated environment within the employees' mobile devices. KNOX ensures the system's integrity when the device is started by implementing a secure boot and at runtime using by a *TrustZone-based Integrity Measurement Architecture* (TIMA). TIMA runs in the secure world of the TrustZone and provides continuous integrity monitoring of the OS by constantly checking the Kernel code and data. Samsung's KNOX is an example of a proprietary approach as its details are unavailable, even to the research community. However, the KNOX Standard SDK is made available via the Samsung Enterprise Alliance Program (SEAP) [102].

2.7 PCAS secured personal device

The PCAS project designed a hardware component called a *Secured Personal Device* (SPD) [103, 104, 94]. The PCAS SPD is a secure hardware storage device that works when connected to a smartphone using USB. The SPD provides a kind of TEE, physically isolated from the smartphone it is connected to. Currently, a few prototypes have been implemented, but it is not yet a commercial product. In this section, we presents background information on the SPD as this thesis proposes security services that can be utilized by the SPD.

The SPD is essentially a system-on-chip with its own battery. It has a large amount of memory so it can securely store a considerable amount of personal data, e.g., personal medical records. It has also biometric sensors, e.g., a front camera to implement face recognition, to authenticate its owner, in order to authorize access to the stored data on the SPD and enforce secure communication with service providers in the cloud when accessing and uploading data. The SPD has no network interfaces so it cannot connect to the Internet directly; instead it uses the smartphone's communication services (e.g., 3G or 4G cellular interface or Wi-Fi interface) as a gateway. The SPD has no screen, only a few LEDs.

The SPD is designed to allow the implementation of several secure mobile apps. An example app involves personal medical records, which may be stored in the SPD for the user's convenience (e.g., to be able to provide them to different hospitals). Another example is a payment app. In this case, a SPD acts as credit/debit card in transactions to purchase products at a point of sale (POS) terminal or vending machines. For example in a vending machine transaction, the user passes the SPD connected to her smartphone in front of an NFC reader, rather than using a credit card. Next the SPD asks the user to authenticate herself and to approve the amount to pay. The SPD contacts the backend of the app in the cloud that deducts the amount from the user's account. Finally, after the backend confirms the payment the vending machine releases the product.

Summary

This chapter presented the security issues associated with mobile devices and various security solutions (i.e., software and hardware-based security mechanisms). Many software-based security extensions and enhancements

to Android platform (such as Kirin [8], TaintDroid [52], Saint [58], QUIRE [44], and IPCInspection [42]) have been proposed to address the security problems of the platform. These proposals present MAC mechanisms at the middleware level that are tailored to the specific semantic of the addressed problem, for instance, establishing a fine-grained access control to privacy sensitive data. These MAC mechanisms can be categorized according to the type of the improvement over the default Android permission framework, including revising the current frameworks and proposing new frameworks. The first category involves the enhancement of the reference monitor module in the Android middleware. The second category provides a new Android permission model to replace the current one. Both improvements are implemented only at the middleware layer of Android's abstraction layer. Thus, they cannot provide a system wide security framework that operates on both the middleware and kernel layer.

Extensions such as TrustDoird [63], FlaskDroid [64], and SEAndroid [67] extend Android with MAC on both the middleware and the kernel layer. They address many security issues of the Android platform which target either the middleware or the kernel layer. However, the existing software-based security mechanisms are inadequate to address attacks that compromise the mobile OS, for instance, if attackers manage to obtain full supervisor privileges.

To secure mobile services despite mobile OS compromise, researchers have proposed hardware security solutions such as MTM [80] specified by the TCG [77] and ARM TrustZone [83, 84]. ARM TrustZone follows a different approach to provide mobile platform security, by extending platforms with hardware supported security extensions. The key foundation of ARM TrustZone is the introduction of both a secure and a normal operating modes into TrustZone

enabled processor cores to provide an isolated execution environment. ARM TrustZone is commonly available in modern smartphones.

Chapter 3

Trusted Posture Assessment for Mobile Devices

Most uses of TrustZone in the literature (as seen in Chapter 2) are based on *measurements* of the normal world, i.e., on hashes of the software running in the normal world obtained using a collision-resistant hash function [97, 98]. This way of using TrustZone is interesting, but the versatility of ARM TrustZone suggests it is possible to obtain richer information about the normal world than just hashes, which are simply numbers with limited semantics. An approach is to analyze the *posture* or *compliance* of the device. The notion of posture assessment was introduced in RFC 5209 [105] for *network access control* [106], which proposed having a software agent running on endpoint devices (such as laptops and desktops) to evaluate and report the posture/compliance of the device to the network owners (e.g., anti-virus software running on the device or not, updates installed or not). The network owner has validation software that determines the device's compliance with the security policies, allowing it to connect to the company's network, to block it, or to connect it to some lower trust virtual local area network (VLAN) (e.g., one that connects only to the Internet).

This chapter presents the design and implementation of DroidPosture, a *posture assessment service* for mobile devices. This service aims to securely evaluate the level of trust we can have on a device (assess its posture) even if the mobile OS is compromised. DroidPosture runs in the devices (e.g., smartphones) and evaluates the security status of their OS (e.g., Android) and apps. DroidPosture is protected from the OS, apps, and malware by leveraging the TrustZone extension to run in the secure world. DroidPosture does introspection of the normal world and can be configured with a variety of assessment mechanisms, e.g., static analysis of apps and detection of rootkits. Posture data obtained with DroidPosture can be sent to *external service providers* such as financial and healthcare institutions, which can use it to decide if they will provide their service to the device or not (or under what conditions they provide it). We also present a communication protocol for this purpose. DroidPosture implements two classes of assessment mechanisms – app and kernel analysis mechanisms – and provides two example of each using: signature-based detector, learning-based detector, syscall table checker, and kernel integrity checker. These mechanisms illustrate the forms of posture analysis that can be implemented in DroidPosture, although others may also be used.

3.1 DroidPosture architecture and design

DroidPosture gives external service providers a mechanism to evaluate the posture of a smartphone and restrict access to critical data based on posture. DroidPosture is a software component that runs in the secure world of a mobile device. The posture information is requested by external services when smartphone apps request to use a service or by a local app to understand the posture of the smartphone.

3.1.1 Use cases

In this section, we describe two use cases for DroidPosture in order to help understand how it can be used.

In the first scenario, an employee of an enterprise that has adopted Bring Your Own Device (BYOD) uses his or her own smartphone to connect to the corporate network and access corporate data (the enterprise is the external service provider). This scenario is quite realistic and a frequent case today in large companies. This scenario is essentially an example of network access control [105, 106], except that we consider mobile devices instead of laptops. The component running in the device is DroidPosture and it is protected using TustZone. The enterprise's security policies restrict access to its assets, for example, requiring adequate posture/compliance of the smartphone to ensure that this device conforms to these policies. These security policies may include various combinations of data, such as location of the device, OS version, device information (serial number, model, open ports, etc.), and security status of the smartphone. The posture assessment involves four actors: the employee, the enterprise, the smartphone app that allows the employee to connect, and the DroidPosture service. The employee sends an access request to the enterprise via the app. The enterprise communicates with the DroidPosture service and requests the smartphone's posture. The DroidPosture service sends a signed block of posture data to the enterprise. The enterprise checks the compliance of the smartphone against its security policies and, if needed, sends remediation instructions for the employee to fix their smartphone in order to bring it into compliance with the enterprise's policies.

In the second scenario, a bank's customer uses a mobile banking app on their smartphone to do a transaction. Depending on the sensitivity of the data, the

bank, which is the external service provider, requires the posture information to determine whether the data passed to the smartphone could be compromised by malware residing in the smartphone. This mobile transaction scenario involves four actors: a customer, a bank, a bank app, and the DroidPosture service. To start the transaction, the customer sends a service request to the bank via the bank app. In order to ensure the security of privacy-sensitive data, the bank communicates with the DroidPosture service via the bank app to request posture information. DroidPosture performs a posture assessment, sends the signed posture to the bank, and the bank sends the data (or not).

3.1.2 Threat model and assumptions

We assume that DroidPosture runs in an ARM processor with TrustZone. We assume that in the normal world the mobile OS and the apps it executes are untrusted, i.e., that they may be malicious or compromised by malware or hackers. In contrast, we assume that the software running in the secure world, including the DroidPosture software, is trustworthy. In order to reduce the size of the TCB [107], the size of the software executed in the secure world has to be as small as possible. Specifically, we use a small tailor-made kernel (see Section 3.2), do not install unnecessary libraries, and have no network stack. The size of the API is also as small as possible to reduce the attack surface, and all inputs are validated. These measures make software attacks against the secure world unlikely to be successful, so in this paper we assume they are not. We also assume that the device (i.e., the normal world) is not yet compromised when DroidPosture is first installed (for example, it can come pre-installed on the device).

Each DroidPosture instance in a device has an identifier id and a public-private key pair (Ku, Kr) for some public-key cryptographic scheme (e.g.,

RSA) [108]. It also has a certificate containing the public key, signed by some trusted certification authority (CA). We also assume the existence of a collision-resistant hash function (e.g., SHA-256) [108].

3.1.3 Architecture

Figure 3.1 represents a mobile device running DroidPosture and communicating with some external service. The *normal world* runs the usual mobile device software: a mobile OS and apps. The normal world also includes a driver (`TZ_Driver`) that allows software in the normal world to call functions in the secure world (in our case DroidPosture). This driver allocates a shared memory zone that is used for the app to pass inputs to DroidPosture, and for DroidPosture to return outputs to the app (in our case these are the assessment results).

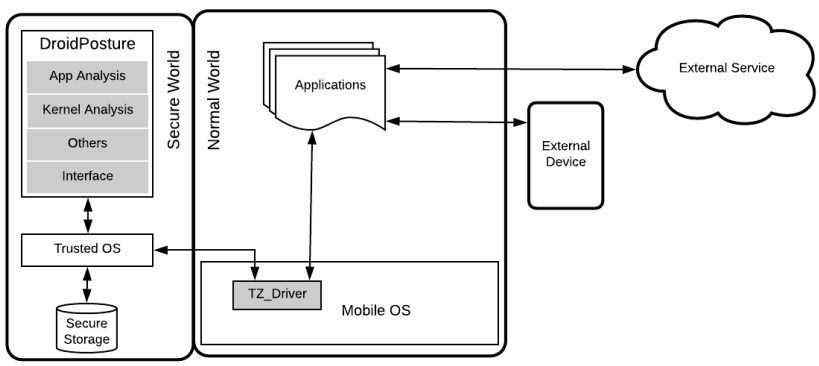

Figure 3.1: Architecture of a mobile device running DroidPosture. The grey boxes are components of the DroidPosture service.

The *secure world* runs the DroidPosture service. This world includes a small trusted OS that provides basic functions for software running in that world (processes, file access, etc.). Besides its private memory space, it is also configured to have a private persistent storage partition (either part of the

device's internal memory or part of an SD card). DroidPosture itself is composed of three modules (*app analysis*, *kernel analysis*, and *interface*), for detecting and reporting the posture of the normal world. The *app analysis* and *kernel analysis* modules are implemented to analyze the posture of the normal world, although others modules may be designed and used (indicated as *others* in the figure). The *interface module* acts as an interface between the normal world and DroidPosture. This interface module receives, validates (for protection against buffer overflows and other input attacks), and replies to requests for posture data from the normal world. Moreover, this interface module collects the posture data from the *app analysis* and *kernel analysis* modules and signs it.

The bootstrapping of the device starts by running the kernel of the secure world, so this kernel is the *static root of trust for measurement* [79]. Before passing the control to the normal world and starting the execution of the normal world, the *kernel analysis* module calculates and stores a hash (measurement) of the normal world OS. This is process is a *trusted boot* [79] and may involve storing hashes of other modules, if needed.

3.1.4 Posture reports

DroidPosture provides information about the posture of the device in the form of a *posture report*. The format of a posture report is: $\langle id, posture\_data, nonce \rangle_{S_{Kr}}$, where *id* is the identifier of the DroidPosture instance, *posture_data* the posture data itself, *nonce* a nonce (a random number used only once for replay protection) that comes with the request for posture data, and S_{Kr} a signature obtained using the DroidPosture instance's private key.

Posture reports can be delivered to apps or transferred to external services via the normal world. In both cases an app running in the normal world requests

posture data by contacting the *interface* module. This request contains the above-mentioned nonce. The module then invokes the *app analysis* and *kernel analysis* modules to collect the posture of the device. When the module gets the result(s), it creates and signs the posture report, and sends this posture to the app, which may optionally send it to the external service.

As the mobile OS and the apps may be compromised, we do not trust them to deliver the posture report unmodified to the app or external service that requested it. The authenticity and integrity of the report are verified using the digital signature S_{Kr} calculated using the private key of the DroidPosture instance in the device. An attacker might still do a denial of service attack by deleting or modifying all posture reports, but this would be understood by the service provider as consequence of a compromised device.

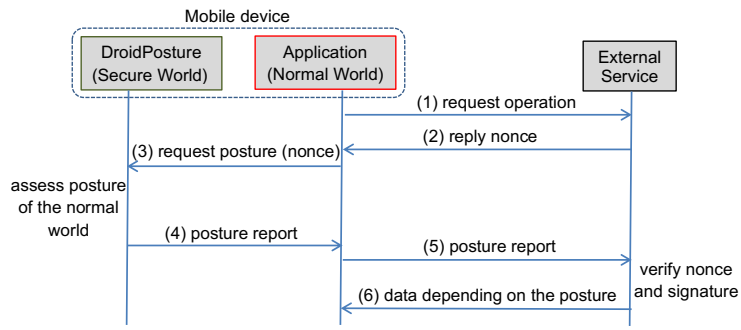

Figure 3.2: DroidPosture providing a posture report to an external service.

Figure 3.2 illustrates the steps for providing a posture report to an external service. An app starts the interaction with the external service, e.g., the backend of the app (step 1). The external service replies and provides a nonce (step 2). The app forwards the nonce to DroidPosture in the secure world and asks for a posture assessment (step 3). DroidPosture performs the posture assessment, creates and signs the posture report, then sends this report to the app (step 4). The app sends this report to the external service (step

5). Finally, the external service verifies the nonce and the signature, using the certificate of that DroidPosture instance (Section 3.1.2). If they are correct, it then interprets the posture data. If it finds the posture acceptable it continues to interact the app, e.g., sending it some data (step 6).

3.1.5 App analysis

This section is about the *app analysis* module (Figure 3.1). This module provides two static analysis mechanisms to detect the presence of malware in Android apps: *signature-based detector* and *learning-based detector*.

By static analysis we mean analysis of code, without executing it. The app analysis module first unpacks the app APK and obtains the bytecodes. Then, the bytecode file is passed as input to the signature-based and learning-based detection mechanisms. Prior to this process, the hash of the APK file is compared with the hashes of the APKs already analyzed stored in the secure world persistent storage partition, in order to avoid re-executing the analysis. If it has already been analyzed, the result obtained previously is returned. Notice that if the APK changes, then its hash also changes (due to the collision resistance property of the hash function).

3.1.5.1 Signature-based detector

The first and simplest malware detection technique is based on pattern matching. Malware detectors have a database of distinctive patterns – *signatures* – of malicious code and they look for them in apps. Malware has to be public for a period of time so that signatures can be generated for that specific malware family.

Our *signature-based* mechanism detects malicious apps or injected malicious code based on similarities of control flow graphs (CFGs). A CFG represents the

control flow of a program. The signature-based mechanism takes the bytecode file and converts each function in the bytecode into a string that represents the CFG of the function. The comparison of similarity of the CFGs is done by using a similarity algorithm such as Kolmogorov distance and normalized compression distance (NCD). The CFG string of each function is compared against the CFGs (the signatures) of known malware in the database to verify if they are similar.

3.1.5.2 Learning-based detector

As the *signature-based* detection mechanism only detects malware for which it has signatures, we propose a complementary mechanism to detect malware in apps. The *learning-based* detection mechanism relies on a machine learning classifier. This mechanism has two phases. In the first phase, *training*, the mechanism statically examines and extracts selected features from known malware samples and benign apps to build feature vectors; then, these features are used to train a machine learning classifier to distinguish malware from normal code. In the second phase, *detection*, the classifier is used to check apps for malware. Notice that the detection phase is executed by DroidPosture itself; while the training phase is executed beforehand, by whoever manages the DroidPosture service.

The selection of features is essential for the efficiency of the detection mechanism. Redundant or relevant features may present several problems such as misleading the learning algorithm, and increasing model complexity and run time. We use the features described below, extracted from the `AndroidManifest.xml` file and the `.dex` file. All features are binary, i.e., either the app has the feature or not:

Requested permissions. Android uses permissions for restricting access to the device resources. Permissions are granted by the user during app installation, or later in the latest versions of Android. Malicious apps request certain permissions (e.g., `SEND_SMS`, `READ_SMS`) more often than benign apps. Requesting such a security sensitive permission is a feature.

Sensitive function calls. Among thousands of Android API functions, we consider API calls invoked by apps that allow access to sensitive data or resources. For example, APIs for accessing the user's personal information, network details, device ID, and sending SMSs.

Suspicious intents. An intent is an abstract description of an operation to be performed. App components are activated using intents. We consider intents that perform sensitive actions as features (e.g., `android.intent.action.CALL`, `android.intent.action.DIAL`).

Suspicious content URI. A content URI is used to locate data in a content provider. It can be used to leak user's personal data or to access another app's data. For example, `content://sms/inbox` can be used to read SMS messages from inbox.

Arbitrary code execution. Execution of native code using JNI or Linux commands. For example, `Ljava/lang/Runtime;->exec()` executes command `exec()` in a separate process.

In the training phase, to construct feature vectors, we retrieve the selected features from each malware sample and benign app and store them as binary numbers, 1 or 0, respectively for presence or absence of the feature. Furthermore, we assign a class to each feature vector, M for malware and B for benign app. Feature vectors are then provided to a machine learning classifier.

We use the *k-Nearest Neighbours* (kNN) algorithm as the classifier [109]. Given vectors of N classes as training samples, kNN classifies an unknown sample by searching the entire set of training samples for the k nearest samples based on a distance metric, then the unknown sample is assigned to the class most common among its k nearest. Since k is a positive integer and if $k = 1$, then the unknown sample is simply assigned to the class of the single nearest sample.

3.1.6 Kernel analysis

This section presents the *kernel analysis* module of Figure 3.1. As stated earlier, this module provides two kinds of analysis.

A rootkit is a piece of malware that gains privileged access to a system, hides itself from the user and the OS, then stealthily carries out some kind of malicious activity. Rootkits may be roughly classified in two classes: (1) *User-level rootkits* replace system binaries and libraries with customized versions and (2) *Kernel-level* rootkits modify the kernel, for example by adding code into the running kernel memory image (`/dev/mem`) or by injecting a Loadable Kernel Module (LKM).

The *kernel analysis* module starts by calculating a hash of the normal world kernel and by comparing it with the hash obtained during the boot of the device (Section 3.1.3). If the hashes are different, the kernel analysis fails immediately (i.e., reports the kernel is compromised).

3.1.6.1 Syscall table checker

Android contains a modified version of the Linux kernel. The Android kernel provides system calls (*syscalls*) that allow apps in user mode to interact with the kernel, usually not called directly but through a library like *glibc*. The Android

kernel provides a set of a few hundred syscalls. Examples are syscalls to perform file operations (`open`, `read`, `write`, `close`), process operations (`fork`, `exec`), and network operations (`socket`, `connet`, `bind`, `listen`, `accept`). Syscalls are one of the primary targets for kernel-level rootkit writers. The kernel uses a *syscall table*, an array of pointers mapping each syscall number to the corresponding function in kernel memory. Modifying a syscall table entry is a popular way to intercept the execution flow of any system service. Kernel-level rootkits often modify syscall table entries to point to new, malicious, system calls. Therefore, in order to detect a kernel-level rootkit, the first step is to verify the integrity of the system call table.

Each time the kernel is compiled, a file containing the map of kernel symbols and addresses is created (`System.map`). Comparing the addresses of syscalls in the `System.map` with the addresses in the *syscall table* during runtime detects if system calls have been redirected, which may be an indication that the kernel has been compromised by a rootkit.

When DroidPosture is installed, our *kernel analysis* mechanism starts by making a copy of the addresses of system calls in `System.map` and storing them in the secure world. Then during runtime the mechanism simply compares that copy with the values in the syscall table in the normal world. Recall that we assume that the system is not compromised when DroidPosture is installed.

3.1.6.2 Kernel integrity checker

Besides syscall table integrity checking, the *kernel analysis* module is capable of checking the kernel code for modifications to detect rootkits. As the kernel is not supposed to change during runtime, changes are probably a sign of malware. For example, a rootkit can replace the first few bytes of some system

call functions with a `jmp` instruction that redirects the execution to malicious code.

In order to verify the kernel integrity, the kernel integrity checker calculates a hash of the kernel code memory pages of the Android OS running in the normal world and compares it against a hash calculated when the system was in a pristine state, which is stored in the secure world persistent storage partition. To calculate a hash value, the start address and length of the target memory pages are required. The kernel integrity checker finds the virtual address of the kernel code in the copy of the `System.map` file stored in the secure world and translates this address to the secure world address space before evaluating the hash value.

3.2 Implementation

This section describes the implementation of a prototype of DroidPosture on an i.MX53 Quick Start Board (QSB) development board. The board is equipped with a Cortex-A8 single core 1 GHz processor, 1 GB DDR memory, and a 4GB MicroSD card. Unlike most TrustZone-enabled smartphones, the i.MX board places no restriction on the use of the secure world.

3.2.1 Runtime environment

Genode is a framework for building special-purpose OSs [110]. It provides a collection of small building blocks (e.g., kernels, device drivers, and protocol stacks). Since requirements vary, Genode can reduce system complexity for each security sensitive scenario. Due to its ability to generate a small TCB, Genode is an appealing foundation for an OS designated to run on the secure world.

The starting point of our prototype was Genode's TrustZone Virtual Machine Monitor (VMM) implementation. In the secure world, we implemented DroidPosture on top of a small kernel based on a custom kernel (*base-hw*) provided by the Genode framework. In the normal world, we run the Android kernel, which is patched to issue hypercalls instead of directly accessing certain resources (such as hardware, persistent storage, and memory) that are preserved for the secure world.

We implemented a driver (the `TZ_Driver` in Figure 3.1) in the kernel for an app in the normal world to issue a hypercall to exit the normal world and trap into the secure world, using the SMC instruction. This driver creates a shared buffer in the normal world RAM that allows data to be passed between the two worlds. Some of the general purpose CPU registers are used to store information about the shared buffer when passing data between the two worlds, including its address and length.

In order to store sensitive data in a persistent way, a part of the SD card or internal memory has to be accessible exclusively by the secure world. We use the Genode partition manager (`part_blk`) for this purpose. It supports partition tables such as MSDOS and GPT, and provides a block session for each partition on a SD card. This allows the partitions to be addressable as separate block sessions and makes it is easy to grant or deny access to them.

In the secure world, it is possible to request the normal world's RAM using an IOMEM session, normally used for memory-mapped I/O regions of devices. The memory is mapped as uncached in the secure world's address space, so DroidPosture modules running in the secure world can access the whole normal world's memory.

3.2.2 DroidPosture modules

Table 3.1 shows the code size of each module implemented in the DroidPosture service.

Table 3.1: Lines of code for the DroidPosture modules.

Modules	**Code Size (LOC)**
App Analysis	30484
Kernel Analysis	142
Interface	207

In our prototype we used components written in Python, which required installing Python 2.6 in the secure world using the Genode libports repository. This is undesirable because it increases the size of the TCB. However, this is not a limitation of our proposal, but of the current prototype. DroidPosture itself does not need to use Python code.

The *app analysis* module is based on Androguard [49], an open source tool written in Python. It is able to unzip an APK file, obtain its metadata and bytecodes. Androguard has a module to create the CFG for each function in a bytecode file. In addition, Androguard has several built-in signatures that are able to detect known malicious apps. Since Androguard is a complete feature-rich framework, we use its modules to disassemble an app's Dalvik bytecode, then create a CFG for each function, and compare these CFGs with the malware CFGs (the signatures) that are stored in the secure world's persistent storage partition. In addition to Androguard, we modified Androwarn [111] to extract the features (Section 3.1.5.2) from malware and benign apps to build feature vectors for the learning-based detector.

3.3 Performance evaluation

To evaluate the performance of DroidPosture, we used a set of micro- and macro-benchmarks by considering calls to the DroidPosture service that: (i) return immediately (baseline); (ii) do app analysis, only signature-based; (iii) do app analysis, only learning-based; (iv) do app analysis, both mechanisms; (v) do kernel analysis, only syscall table checker; (vi) do kernel analysis, only kernel integrity checker; (vii) do kernel analysis, both mechanisms; (viii) do all the detection mechanisms.

In the *micro-benchmarks*, an app (in the normal world) sends a request for posture and gets a reply back from DroidPosture (in the secure world). The *macro-benchmarks* are used to evaluate the posture assessment transmission protocol. For this purpose, we used a remote server which runs on a standard laptop. The server listens for incoming requests from the app in the normal world and sends requests for posture to the DroidPosture service running in the secure world of our board via the app.

3.3.1 Micro-benchmarks: mechanism performance

We used the calls mentioned above to evaluate the overhead of DroidPosture. To measure the time for the baseline (i), the app in the normal world sends a request for posture to the DroidPosture service in the secure world that does not execute any analysis module. We repeated the experiment 1000 times and obtained an average of 0.082 *ms*, with standard deviation of 0.0061 ms.

For the rest of the calls the process is similar, except that DroidPosture executes a subset of the analysis modules. We expected calls to the app analysis modules to depend on the size of the apps, so we considered a set of apps with different sizes (downloaded from Google Play Store). We did

experiments for the combinations of calls (ii) to (viii) and all bytecode sizes. The results of these experiments are shown in Table 3.2. Moreover, Figure 3.3 presents the same values, but only for the combinations of app analysis modules and the total.

Table 3.2: DroidPosture delay when called locally (in seconds).

Size		**Calls**						
APK	**.dex**	**ii**	**iii**	**iv**	**v**	**vi**	**vii**	**viii**
12KB	5KB	1.81	0.8	2.23	0.15	1.64	1.75	4.01
19MB	39KB	14.12	1.51	14.92	0.14	1.63	1.77	16.26
4MB	67KB	31.84	1.57	32.26	0.14	1.63	1.77	33.19
250KB	103KB	29.03	1.55	29.70	0.14	1.63	1.77	30.40
803KB	153KB	66.28	5.85	69.96	0.14	1.63	1.77	71.21
401KB	305KB	206.21	7.86	206.54	0.14	1.63	1.77	208.42

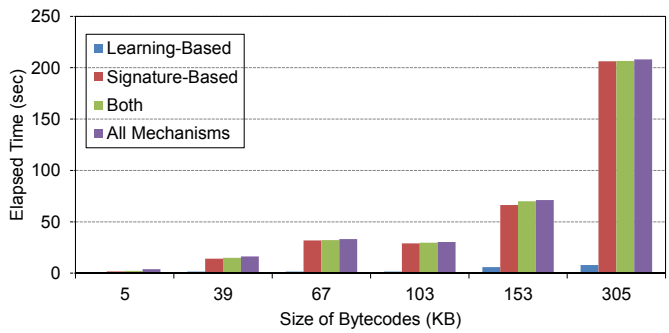

Figure 3.3: DroidPosture delay when called locally with emphasis on the apps analysis modules (in seconds).

These results allow us to extract several conclusions. First, in the table it is clear that calls to the kernel module have a delay that is *independent* of the size of the app, as expected (columns *v-vii*). Second, both the table and the figure show that delay of the signature-based analysis grows with the size of the `dex` file, to the point of becoming unusable (column *ii*). This was expectable as it converts all the functions in the bytecodes into CFGs, which increase with the size of the code. Third, the table and the figure also show

that learning-based analysis grows slowly with the size of the `dex` file, showing that this form of analysis is much simpler and faster than the signature-based (column *iii*). Fourth, they also show that these two delays depend on the size of the `dex` files, not on the size of the APK files, which often contain many files that are not analyzed, e.g., images and video (columns *ii*-*iii*). Fifth, all mechanisms and their combination seem to be usable, except the form of signature-based analysis we considered.

3.3.2 Macro-benchmarks: DroidPosture in a company

To evaluate the performance of the DroidPosture service in the context of a realistic use case, we measured the total time for the remote server to send a request for posture and to get a reply back from the service (see Figure 3.2, page 53). We used a LAN network to emulate the case of posture being provided inside a company.

We measured a round trip time (RTT) between our board and the remote server of 0.497 *ms*. We used the same calls as before. In this case, the time for a baseline call was 1.92 *ms*, with standard deviation of 0.096. The results of these experiments are shown in Table 3.3. The trends are essentially the same as were observed with the micro-benchmarks, with the additional delay of the network.

Table 3.3: DroidPosture delay when called by a remote service (sec.).

Size		**Calls**						
APK	**.dex**	**ii**	**iii**	**iv**	**v**	**vi**	**vii**	**viii**
12KB	5KB	1.85	0.89	2.29	0.17	1.67	1.78	4.59
19MB	39KB	14.27	1.56	14.94	0.16	1.65	1.78	16.74
4MB	67KB	31.86	1.60	32.38	0.16	1.65	1.78	34.05
250KB	103KB	29.07	1.57	29.77	0.16	1.65	1.78	31.24
803KB	153KB	66.39	5.87	69.12	0.16	1.65	1.78	72.32
401KB	305KB	206.61	7.88	207.11	0.16	1.65	1.78	208.89

3.4 Security evaluation

As previously mentioned, the specific modules we implemented in DroidPosture serve mainly to demonstrate the kinds of analysis it can make and that it can support several. Nevertheless, we evaluated experimentally the quality of the detection made by our four modules, which we present here.

We used 500 malware samples from the Drebin dataset [50]. This dataset contain samples from 179 different malware families collected between August 2010 and October 2012. We balanced the number of samples from different malware families. For benign apps, we randomly downloaded 30 apps from 8 different categories on Google Play Store and verified them through VirusTotal that runs samples through around 10 anti-virus products, in order to get some confidence that they had no malware (`https://www.virustotal.com`).

To evaluate the detection performance of the *learning-based detection* mechanism, we randomly split our datasets into a training set (66%) and a test set (33%). The training set was used to determine the classification model, whereas the test set was used for measuring the detection performance. We use as metric *accuracy*, which evaluates the ratio of apps correctly classified (it is given by the number of apps correctly classified as good or bad, divided by the total number of apps evaluated). The result shows that the learning-based mechanisms using *kNN* with $k = 3$ had accuracy of 89.4% with a false positive rate (i.e., percentage of samples wrongly identified as malware) of 4%. The detection performance is relatively good, although our dataset is not large. This suggests that our features effectively model malicious code.

The *signature-based detector* achieves better detection performance for samples that have signatures in the database. To test its performance, we created signatures from over 100 different malware families, such as DroidDream,

DroidKungfu, DogoWar, and foncy. The signature-based detector was able to detect malware samples from those malware families correctly with approximately 100% accuracy. However, the learning-based mechanism is more effective than the signature-based mechanism for apps that contain unknown malware.

To illustrate the effectiveness of *kernel analysis* modules, we deployed the Mindtrick kernel-level rootkit on our board [112]. The Mindtrick rootkit replaces the entry for the read syscall (`sys_read`) to instead point to the address of a malicious function injected into the kernel. It allows attackers to obtain a reverse TCP shell on Android devices. Our kernel analysis module in the secure world is able to detect this rootkit by reading each address in the system call table from the normal world memory and then comparing it with each syscall address listed in `System.map`. It correctly inserted an error for the `sys_read` syscall entry in the posture report.

3.5 Related work

Introspection has been proposed in the context of PC virtualization with hypervisors as a solution to protect anti-malware and intrusion detection mechanisms from malware and hackers [113, 114, 115]. The idea is to run these mechanisms in a thin virtual machine (VM) isolated from the VM that runs the OS and the apps. DroidScope [18] enables introspection for Android smartphones to monitor the behaviour of app at different layers of the platform, such as OS and Dalvik VM. Yan and Yin applied this approach to detect malware in Android smartphones using a customized QEMU hypervisor [18]. However, the size and complexity of hypervisors still make them a target for malware, similarly to what happens with OSs. Moreover, hypervisor-based malware analysis techniques increase power consumption, reducing the

battery operating time on smartphones. In contrast, we take advantage of the ARM TrustZone to provide a hardware-assisted isolated environment to protect a posture assessment mechanism that aims to detect and report the trustworthiness of Android devices by analyzing the apps installed on the device and the kernel.

As seen in Chapter 2, several mechanisms use ARM TrustZone to associate measurements of the normal world with specific data, e.g., login data and sensor readings [98, 97]. However, a vector of hashes of a few bytes provide little information about the state of the device. In this work, we used the TrustZone to protect a posture assessment mechanism that aims to detect and report the trustworthiness of Android devices by analyzing the apps installed on the device and the kernel.

3.6 Summary

This chapter presented the DroidPosture service, which is protected by the ARM TrustZone extension. The service aims to securely detect intrusions in an Android device and report posture information for external services. We implemented a set of app and kernel analysis mechanisms to exemplify the kind of posture assessment that our service can do, although the specific analysis to do are probably specific to different scenarios. The performance of these mechanisms seems to be adequate for many apps, with the exception of the signature-based analysis that is slow for large apps.

Chapter 4

Dynamic Analyser for Android Apps

This chapter presents the design of the *Trustzone-based Trace analyser for anDroid apps* – T2Droid. This mechanism does dynamic (runtime) analysis of apps to detect malware on Android-based mobile devices (in contrast with DroidPosture that only did static analysis). T2Droid uses traces of Android API function calls and kernel system calls (syscalls) performed by an app to detect whether it is malicious or not. This combination of the two types of calls allows observing operations with a clear semantics (e.g., sending an SMS message), while not letting malware escape this detection by running native code and doing syscalls instead of calling API functions. It uses a machine learning classifier to do the detection, which allows it to be configured without a human to manually develop detection rules, and to be reconfigured easily when new malicious apps are discovered. T2Droid is protected from malware by leveraging the TrustZone extension. The detection itself is performed inside the secure world. The capture of the API function calls and syscalls has to be done by software components running in the Android kernel, in the normal world, but there is a protected component in

the secure world that verifies the integrity of these normal world components and of the mobile OS kernel.

T2Droid does not aim to substitute static analysis mechanisms that should be used to test an app before it starts being distributed in an app marketplace [116]. Instead, it is a complementary mechanism that provides a second layer of protection at runtime, similarly to anti-virus software. This second layer of defense is important, as many apps distributed in marketplaces are malicious [117, 3]. However, unlike anti-virus software, T2Droid is protected from malware by leveraging the TrustZone extension.

We envisage three main use cases for T2Droid. The first two consider personal mobile devices, typically smartphones. The first is to run T2Droid automatically whenever an app is downloaded from a marketplace and installed. The first time the app is executed, T2Droid would be executed during a configurable amount of time or number of API calls in order to check if the app is trustworthy. The second is to run T2Droid when requested by the backend of the mobile app, i.e., by the part of the app that runs in the cloud or company servers. The objective would be for the backend to assess if the app in the mobile device has been compromised. The third would be to run T2Droid in devices targeted specifically at testing the trustworthiness of apps. T2Droid would run when the app starts to run, but would be executed for a larger period of time in order to check the app for a longer period.

4.1 T2Droid architecture and design

This section presents T2Droid's architecture and design.

4.1.1 Threat model and assumptions

T2Droid runs in an ARM processor with TrustZone. We make the same assumptions about the software running in the normal and secure world, the TCB, and the API to the secure world as were stated in Section 3.1.2 on page 50. Malware or attackers might be interested in disabling T2Droid, as they do to anti-virus and other anti-malware software, but (1) we assume they cannot compromise the part of T2Droid that runs in the secure world and (2) we use code in the secure world to verify the integrity of the components that run in the normal world. We assume the existence of a collision-resistant hash function (e.g., SHA-256).

4.1.2 Architecture

The architecture of T2Droid is shown in Figure 4.1. The *normal world* runs the Android OS and apps. The normal world also includes a part of T2Droid, specifically the two *tracer* modules, which obtain information about the behaviour of the app being checked. The *API calls tracer* and *syscalls tracer* are in charge of monitoring Android API calls and kernel syscalls, respectively. These components are tightly integrated with the Android environment, so we place them in the normal world. It would be possible to place them in the secure world, but there are two drawbacks: (1) the implementation would be much more complex and (2) the performance overhead would be high as there would be at least an order of magnitude more context switches between the two worlds.

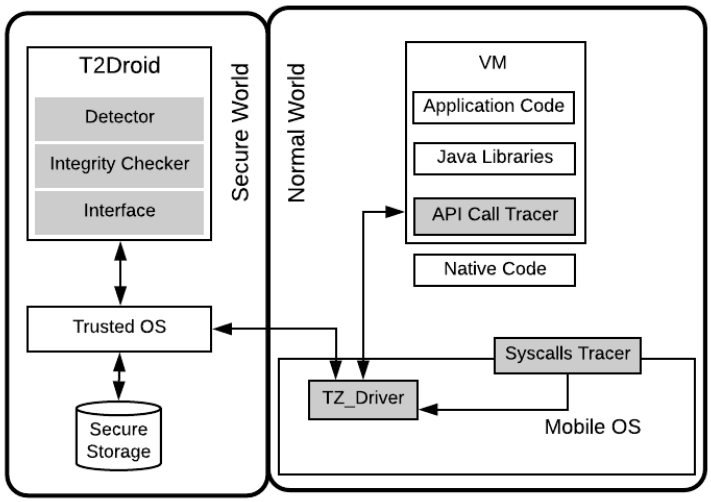

Figure 4.1: Architecture of a mobile device running T2Droid (grey boxes).

The TrustZone driver (`TZ_Driver`) is a kernel level driver, which enables the tracer module to communicate with the secure world. It allocates a shared memory zone that is used for the *tracer* module to pass trace files to the *detector* module in the secure world. It is also used by an app or another module to order T2Droid to inspect an app. For example, in the first use case mentioned above, the order may come from a modified *app installer*, the Android component that installs new apps.

The *secure world* runs a small trusted OS that provides basic functions for software running in that world (processes, file access, etc.) and modules of T2Droid. The *detector* module receives trace data of an app from the *tracer* modules and performs detection using a machine learning classifier. The *integrity checker* module verifies the integrity of the *tracer* modules and the Android kernel running in the normal world. The *interface* module provides an interface for the software in the normal world to interact with T2Droid.

It validates all the incoming data from the normal world and protects against buffer overflows and other input attacks.

Next we explain each component of the architecture.

4.1.3 Tracers

T2Droid analyses the behavior of an app by observing the calls it makes. Tracers extract sequences of API calls and syscalls. Next we present these two components, starting with the *API calls tracer*, then the *syscalls tracer*.

4.1.3.1 API calls tracer

Android apps rely heavily on middleware-layer libraries, i.e., they frequently call their APIs. Access to these APIs is protected using Android's permission framework, but users are compelled to give the permissions requested, otherwise they cannot use the app. The sequence of API calls performed by an app reveals to some extent its behavior. Malicious apps often make calls that have legitimate uses, but may be associated with malicious behavior. Examples include sending SMS messages, making phone calls, or accessing the user's contacts. Therefore, analysing Android API function calls is a way of detecting malicious behavior.

Dynamic analysis requires apps to be instrumented with inspection code. There are two main instrumentation approaches. *Static instrumentation* involves modifying the app's APK file before the app is installed or executed. With *dynamic instrumentation*, the code is injected into the app's process memory by an external process while the app is being launched. This approach does not require modifications to the APK file that might cause reliability issues and would be easier to detect by malware.

For these reasons, the *API calls tracer* module relies on dynamic instrumentation. It runs custom code before and after an Android API function is called. Since every app in Android runs in its own VM, the injected code has access to the VM and is executed inside the VM to hook and call selected Java API methods of the target app. The number of API calls available in Android is large, so it is convenient to limit tracing to a subset of these calls based on considering whether they are likely to be used by malware.

The tracer is configured with the number of API calls to collect or the time to collect them. When this threshold is reached, the tracer sends the traces to the T2Droid detector module in the secure world.

4.1.3.2 Syscalls tracer

Android apps may contain native code, so malicious apps may use such code to avoid calling Android API functions and perform malicious operations in a way that is unobservable by the *API calls tracer*. However, such code has to call Android, so we observe its behavior by tracing syscalls.

As explained in Section 3.1.6.1 on page 57, syscalls are the fundamental APIs that allow apps to call the OS kernel to perform a variety of operations, such as creating processes, reading from files and network sockets, etc. Therefore, capturing and analyzing the syscalls performed by an app may provide information about accesses to the file system and network, communication with other processes, etc.

The *syscalls tracer* intercepts and logs syscalls being made by a running app using the `ptrace` syscall. T2Droid uses the number of calls to each syscall to analyse the behaviour of the app. Similarly to the *API calls tracer*, the *syscalls tracer* is configured with the number of syscalls to collect, or the period to collect them.

4.1.4 Feature selection

Unlike the previous and the next sections, this section does not present a component of the T2Droid architecture, but rather explains an important aspect of the *detector*.

Features are measurable characteristics of a certain phenomenon and play a crucial role in machine learning. In our case, the features correspond to function calls and characterize the behavior of an app (the phenomenon). Similarly to learning-based detection mechanism in Chapter 3 Section 3.1.5.2, T2Droid aims to classify apps in two classes – malicious or not – based on a machine learning classifier. The selection of which features to use is important for the detection mechanism to give good results.

T2Droid uses a vector composed of two types of features: those related to calls to the Android APIs and those related to syscalls. There is one feature per API function and per syscall. The features of the first set (Android APIs) are binary, i.e., they take value 1 if the app made that call, otherwise they take value 0. The features of the second set (syscalls) take a value that is an integer equal or greater than zero, corresponding to the number of times the syscall was issued by the app. We make this distinction because we assume the number of calls made to one among the large number of API calls is not very relevant, whereas the number of calls made to the lower number of syscalls is. Our experimental results seem to substantiate this assumption (see Section 4.4 starting on page 85). Nevertheless, it would also be possible to use only binary features, or only integer features, or integer features for Android APIs and binary for syscalls.

Among thousands of Android APIs, we identified the sensitive/suspicious API calls as those that are often invoked by malicious apps. We analyzed

a large set of malware and benign apps and generated a list of distinct API calls, then extracted those frequently used by malware. This reduced our features to 121 APIs, which is of the same order of magnitude as the number of syscalls. Our features for Android APIs are a list of these calls in the format: `full-class-name;method`. Some examples are shown in Figure 4.2. In contrast, for the syscalls we considered as features all the syscalls. The name of the feature is the name of the syscall (e.g., `open`, `read`).

```
android.telephony.TelephonyManager;getPhoneType
android.telephony.TelephonyManager;getNetworkOperator
android.app.SharedPreferencesImpl;getInt
java.io.FileOutputStream;FileOutputStream
android.app.SharedPreferencesImpl;getBoolean
java.security.MessageDigest;update
java.io.File;mkdir
```

Figure 4.2: Examples of API call features extracted from a trace.

4.1.5 Malware detection process

This section describes the malware detection process, which mainly involves the *detector* module of the T2Droid architecture (Figure 4.1 on page 72). The *detector* module is essentially a machine learning classifier and the core of T2Droid. It runs in the secure world. In this section we assume the integrity of the *API call* and *syscalls tracer* modules (we defer to Section 4.1.6 an explanation of how this assumption is enforced).

There are three important phases of the life cycle of the classifier to consider. The first phase is the selection and training of the classifier, which we have done and report in this chapter. The second phase is the use of the classifier at runtime, which we validated experimentally. The third is the re-training of

the classifier during the life cycle of T2Droid, which we only briefly explain as it is a repetition of part of the first phase.

The first phase is the selection and training of the classifier. This phase is not done inside the device. We first picked a balanced dataset of malicious and benign apps (see Section 4.3 starting on page 82). Then, we analysed all these apps with VirusTotal, an online malware scanning tool, in order to confirm that they were indeed malicious/benign. Next, we extracted feature vectors from all the apps. These vectors were then provided to a set of machine learning classifiers available in the Weka tool [118], e.g., kNN and SVM, and their detection effectiveness was compared using different metrics. The best classifier was then implemented in the *detector* module and trained with the same dataset (details in Section 4.3).

The second phase is the use of the classifier at runtime to analyse traces. The *tracers* provide the *detector* with traces. To do so, the *tracers* pass traces to the `TZ_Driver`, which passes it to the *interface* module within the secure world. The *detector* module transforms traces into a vector of features. Then the classifier classifies this vector as characterizing a malicious or benign app. Experimental results are presented in Section 4.4.

The third phase consists of re-training the classifier. As malware evolves, we expect the features selected and the training done to become inadequate and the performance of T2Droid to decrease with time. Therefore, the classifier has to be re-trained periodically. Similarly to the training phase, re-training is not done in the device. Re-training involves again selecting the features to be used and repeating the training phase. All of the existing instances of the T2Droid service running in mobile devices will have to be updated securely using a scheme similar to those used by anti-virus software, e.g., using the cloud [119]. If necessary, the classifier may also be changed, but this is more

complicated than updating the classifier configuration as it involves changing its code.

4.1.6 Normal world integrity verification

In Section 4.1.5 we assumed the integrity of the *API call* and *syscalls tracer* modules, although these modules are executed in the normal world. In this section we explain how this assumption is enforced. This enforcement involves three aspects – trusted boot, system integrity verification, and tracer integrity verification – that are implemented by the T2Droid *integrity checker* module (Figure 4.1). This module is further divided into three sub-modules: *boot support*, *system verifier*, and *tracer checker*.

The normal world integrity verification is made at two moments. First, when the device is started, a *trusted boot* is executed. Second, when a malware detection is requested (Section 4.1.5), first a system integrity verification is executed, then a tracer integrity verification is done. If either of these two verifications fails, then the malware detection terminates immediately with failure.

4.1.6.1 Trusted boot

As explained in Chapter 2 Section 2.5.1, the best-known implementation of the trusted boot uses TPM [78]. More recently, the same idea has been implemented using Intel SGX technology.

In our system we implement a trusted boot process. We assume that when the device boots, the secure world is booted first, then it passes control to the normal world, which boots normally starting with the bootloader, which is the most common booting process for devices that have TrustZone, such as i.MX53. Therefore, in our case, the secure world is the SRTM, so it computes

a hash over the (normal world) bootloader. The module in charge of obtaining this hash is the *boot support* module. Then, the normal world bootloader is executed and computes a hash of the Android kernel. The bootloader passes this hash to the secure world *boot support* before passing control to the kernel. To do so it contacts the `TZ_Driver` in a manner similar to calling a remote procedure with the API: `T2Droid_store_measurement(hash_t hash);`

When the Android kernel starts to run, it does a measurement of the *init* program, in charge of initializing several elements of Android, and calls the same function to pass the measurement to the *boot support* module. Then, *init* measures the program *app_process*, which when executed becomes the *zygote* process, i.e., the first instance of the Dalvik VM (the VM that executes all Android apps). Again, *init* calls the function to pass the measurement to the *boot support* module. The secure world does not have PCRs as it is not a TPM, but instead it stores the hashes in a vector in the secure world that plays a similar role to the array of PCRs. Notice that the function does not take any input other than the hash; the hashes are simply stored by the *boot support* module in the order it obtains them.

This process allows later verification of whether the normal world has been compromised.

4.1.6.2 System integrity verification

The *system verifier* module detects whether the normal world has been compromised. The module does measurements of the components measured during the trusted boot process: bootloader, kernel, *init*, and *app_process/zygote*. Then, it compares these hashes with those stored during the boot process. If they differ, then the system must have been compromised due to the collision

resistance property of the hash function, hence the verification fails. Otherwise, the verification passes.

4.1.6.3 Tracer integrity verification

The *tracer checker* is responsible for verifying the integrity of the code of the tracer modules using a hash function (the code not the data, that changes) to validate the traces collected by these modules. When it is called, it calculates a hash `h` of each of the modules running in the normal world. Then, it compares this hash with the hash of the module it keeps in secure storage ($\mathtt{h}_{st}$). The value of $\mathtt{h}_{st}$ comes with T2Droid, it is not obtained during the trusted boot, as it is not part of the boot process. If they match, then the check is successful, otherwise it fails. This check is possible because the secure world can access the resources of the normal world, as previously mentioned.

The tracer modules might be maliciously modified just after the tracer integrity verification. However, the system integrity verification provides assurance that the software running in kernel mode (the kernel itself) is not compromised. However, such a modification would still be possible if there was a vulnerability in the kernel or the tracers. A solution would be to repeat the verification of the tracers multiple times during the capture of a trace; however, this scheme might be attacked using a race [120], but the probability would be much lower.

4.2 T2Droid implementation

We implemented a prototype of T2Droid on an i.MX53 QSB board. We used the same runtime environment that was implemented for DroidPosture (see Chapter 3). The Android kernel running in the normal world has to be modified to obtain the measurements and send them to the secure world. Our current implementation still does not support this feature, but it could be

implemented following a similar mechanism to that implemented in the Linux kernel that sends hashes to the TPM (e.g., the TrustedGrub bootloader).

For implementing the *API calls tracer* module we used the *Xposed framework*, which allows modifying the behavior of Android apps without modifying their code and the APK file [121]. There are alternative frameworks, *Cydia Substrate* and *Frida*, but Xposed seemed to be the most stable, with a support community and frequent updates. All app processes in Android have as their parent a process called *Zygote*, i.e., every app is created as a fork of that process. *Zygote* is the first process started by `init.rc` after the device boots. This process is launched by the `app_process` executable (`/system/bin/app_process`), which loads all necessary classes and resources. Xposed takes advantage of this mechanism and replaces the *app_process* file with a modified one. Whenever a new VM is created, this *extended app_process* adds an additional jar file (`XposedBridge.jar`) to the classpath. Xposed allows hooks to be added to Android API functions and extending them with our own custom code written as a module that is loaded by the *extended app_process* when the target app process is launched. Our *API Calls tracer* is a module of this kind that records the invocation of Android API function calls in a log data structure. The log is later processed and analyzed by the *detector* module in the secure world.

The *syscalls tracer* was implemented based on *strace*, a debugging tool for Linux and related OSs. The *strace* tool can be used to trace the syscalls made by a process. It can be considered to be a user space interface to the `ptrace` syscall. The T2Droid *syscalls tracer* module records the name of each system call, the arguments passed to the system calls and their return values. After the app to be analyzed has been started, a *syscalls tracer* instance is launched and attached to the VM running the app. The above-mentioned data is logged

to a data structure to be processed and analyzed by the *detector* module in the secure world.

The *detector* is the main component of T2Droid executed in the secure world. It receives the traces from the two *tracer* modules and runs the detection algorithm. It was implemented based on the Java code of the algorithm in the Weka tool.

The implementation of the *integrity checker* in the secure world follows what was explained in Section 4.1.6. This module was implemented to have access to the normal world's memory and the Android file system partitions.

4.3 Selection and training of the classifier

This section explains how the detection algorithm was selected and trained. For this purpose, we collected 80 Android malware samples up to 3 years-old (2014-16) from the Contagio mobile repository [122]. These samples belonged to 21 different malware families, e.g., FakeInstaller, DroidKungFu, and Opfake. For benign apps, we downloaded from Google Play Store 10 recent apps selected randomly from 8 different categories. Then, we verified these apps with VirusTotal to ensure that no anti-virus product recognizes it as malware. This gave us a balanced dataset of 160 apps, half malicious, half benign.

To extract the features, we obtained execution traces by executing all these apps. For this purpose, we used Android Monkey [123] to generate different kinds of events for the app. Monkey is a program running on Android provided by the Android SDK, which automatically feeds an app with pseudo-random streams of user events such as clicks, key presses, and touches, as well as a number of system-level events. We executed and traced each of the apps using Monkey to generate 500 events with a delay of 1 second between each pair of

events, leading to more than 8 minutes of execution, which is enough to extract reasonably long traces (100-200 KB for syscalls and 1-3KB for API calls). This time is a tradeoff between how long we monitor the app (the shorter the time the better) and how much of its behavior we observe (the longer the time the better). Due to the complexity of using Monkey and executing these experiments on the board, we executed the apps in the Android emulator [124], set to emulate an ARM CPU.

For each app, we then extracted the features. For the API calls, we extracted 121 values 1 (call issued) or 0 (not issued). For the syscalls, we extracted the number of calls made to each. We assigned a class to each feature vector, M for malware and B for benign app. We created three feature vector sets: (1) API calls only (only features extracted from API calls traces); (2) syscalls only (only features extracted from syscall traces); and (3) all features extracted (both API calls traces and syscall traces). The purpose of having variants with these three sets is to allow comparison of the results and understand if there is a benefit in using more than one of the traces.

The feature vector sets were then inserted in Weka. Weka allows training of different machine learning classifiers with a set of feature vectors and obtaining metrics of their performance. For each feature set, we conducted experiments using six widely used machine learning classifiers: Bayes Net, Naive Bayes, SMO (SVM), Ibk (kNN), J48, and Random Forest. In each experiment, we used 10-fold cross validation to evaluate the classifiers without having a training and a testing dataset. Our sets of apps (both malicious and benign) were divided into 10 different sets/groups. In each of the 10 rounds, one set of malware and benign apps was used as the testing datasets and the remaining 9 as the training datasets.

For each classifier and feature set, we measured five common performance metrics. Consider that *TP* (*True Positives*) is the number of malware samples correctly identified as such, *FN* (*False Negatives*) is the number of malware samples classified as benign apps, *TN* (*True Negatives*) is the number of benign apps correctly identified, and *FP* (*False Positives*) is the number of benign apps identified as malware. We consider the following metrics:

$$Accuracy = (TP + TN)/(TP + TN + FP + FN)$$

$$True\ Positive\ Rate\ (TPR) = Recall = TP/(TP + FN)$$

$$False\ Positive\ Rate\ (FRP) = FP/(FP + TN)$$

$$Precision = TP/(TP + FP)$$

$$Fmeasure = 2 \times Recall \times Precision/(Recall + Precision)$$

The two most interesting metrics are *precision* (*Prec.*), which measures the confidence we can have when T2Droid says an app is malicious that it is indeed so, and *accuracy* (*Acc.*), which measures the correctness of the mechanism in terms of the rate between correct results and the total.

Table 4.1: Evaluation of 6 classifiers with 160 apps and 3 feature vector sets.

	API calls only		**syscalls only**		**API calls and syscalls**	
	Accu.	**Prec.**	**Acc.**	**Prec.**	**Acc.**	**Prec.**
BayesNet	0.84	0.86	0.91	0.91	0.95	0.96
NaiveBayes	0.78	0.78	0.76	0.78	0.85	0.89
SMO (SVM)	**0.98**	**0.99**	0.86	0.87	0.97	0.97
Ibk (kNN)	0.94	0.95	0.91	0.93	**0.98**	**0.99**
J48	0.92	0.94	0.92	0.93	0.86	0.87
RandomForest	0.97	0.97	**0.94**	**0.95**	0.94	0.94

The results of this evaluation are shown in Table 4.1. Comparing the precision and accuracy, it is possible to conclude the following. First, the best classifier with API calls only is SMO, which is an implementation of support vector

machines (SVM). Second, with syscalls only the best classifier is Random Forest, and the results are slightly worse than with API calls. Third, the best performance was obtained with both API calls and syscalls with the Ibk algorithm, an implementation of the *k-nearest neighbors* (kNN) algorithm, with an accuracy of 0.98 and a precision of 0.99, although the accuracy was always 0.85 or greater and the precision 0.87 or greater independently of the classifier used. This lead us to the conclusion that *the detector should use the two types of traces and use Ibk/kNN as the classification algorithm.* Therefore, this was the algorithm implemented in T2Droid.

4.4 Experimental evaluation

This section presents the experimental evaluation of T2Droid in terms of detection and performance.

4.4.1 Detection

Our experimental evaluation of the detection performance of T2Droid was presented in the previous section, together with the study that lead to the selection of the classifier. The experimental results for the selected algorithm (Ibk/kNN) were shown in bold for the case of API calls and syscalls in Table 4.1, showing that it had excellent performance in all metrics, with all equal to 0.98 or 0.99, except FPR that was 0.02 (but in this case values close to 0 are better).

Receiver operating characteristics (ROC) curves are a well-known tool to visualize the performance of classifiers. Therefore, we plotted the ROC curve for the T2Droid detector in Figure 4.3. Moreover, we plotted two curves for detectors with the same algorithm but with API call features only and systcall features only. The ROC curve is obtained by plotting the TPR versus

the FPR with various threshold settings. The figure confirms that Ibk/kNN is indeed a good classifier when both types of features are used, as the curve rises rapidly to values close to 1, then stays there. Moreover, the figure shows that the results when using both types of features are better than the results with API calls only, which are better than syscalls only.

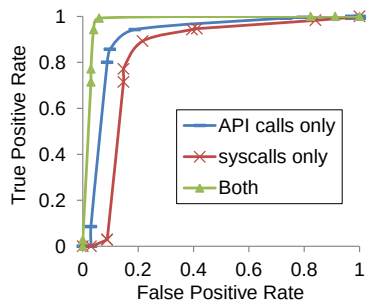

Figure 4.3: ROC curve for the detector in T2Droid (Both) and the detector with only one of the types of traces (the other two lines).

4.4.2 Performance overhead

As mentioned above, we did not measure the time of the whole analysis as the time to extract the traces is configurable. However, we measured the total time required for the tracer modules to send trace data (both API calls traces and syscalls traces) to the detector module. This time includes the performance delay introduced by the context switching, and then copying and sending of data between the two worlds using the shared buffer. In addition, we measured the time for the detector module to prepare a feature vector from the trace data and classify the examined app as malicious or benign (with the kNN classifier). For this, we used Monkey to generate different kinds of events while running the examined app. We repeated this experiment for different numbers of events. The results are shown in Table 4.2. For each number of events, the total time to complete the above operations is shown in the last

column of the table. We show the row for 500 events in bold as this was the case considered in the detection experiments.

Table 4.2: Time for trace transfer, feature vector preparation, and classification.

Num. events	**syscall traces (B)**	**API call traces (B)**	**Trace transf. (s)**	**Feat. vect. prep. (s)**	**Classif. (s)**	**Total (s)**
100	13.2K	488	0.000012	0.082	2.14	2.22
200	69.3K	1.3K	0.000036	0.17	2.21	2.38
500	**157.2K**	**2.1K**	**0.000053**	**0.26**	**2.21**	**2.47**
1000	252.9K	2.6K	0.000065	0.34	2.23	2.57
1500	531.8K	4.4K	0.00011	0.62	2.22	2.84

Table 4.3: Time to do integrity verification.

File name	**Size (KBytes)**	**Time (ms)**
`XposedBridge.jar`	115	71.62
`app_process`	22	13.54
`init`	90.1	48.71
API calls tracer module	2909	2511.21
syscalls tracer module	1126	829.55
Android kernel	8324	932.66
Total	*12586.1*	*4407.29*

We also evaluated the performance overhead incurred by the normal world integrity verification process. The *system verifier* checks the integrity of the normal world by calculating hashes of the Android kernel, init, and app process using SHA-512, and compares them against their known-good values. The *tracer* checker also does the same operations for the API call and syscalls tracers running in the normal world to verify their integrity. The times for these operations are shown in Table 4.3. The results are the average of 1000 repetitions. The table shows both the size and the time needed to check the integrity of the modules. For the *API calls tracer*, we show separately

the values for the module provided by the *Xposed* framework, as it is also important to ensure the integrity of the system: `XposedBridge.jar`. The last line shows the total. The total time required to check the integrity of the normal world is around 4.4 seconds in our board.

4.5 Related work

Several dynamic analysis mechanisms have been proposed to control information flow in real-time, e.g., in order to prevent the flow and exposure of privacy-sensitive data [52, 53, 14]. This is an interesting approach but requires human knowledge about the ways malware violates security properties, whereas in T2Droid we utilize machine learning to extract such knowledge automatically. Moreover, the methods used to control information flow in real-time require modifications to Android, which is something we want to avoid.

Another trend in dynamic analysis is detecting malware by evaluating calls [54, 125, 21]. An issue is the semantic gap between syscalls and high level behavior, such as sending an SMS message. An alternative that greatly reduces this gap is to trace Android API calls, which is the approach followed in some recent work [11, 12, 13]. T2Droid follows this trend but evaluates both syscalls and Android API calls. Moreover, T2Droid is a system, whereas existing work proposed only detection approaches based on machine learning.

As mentioned before, in terms of security these dynamic analysis mechanisms have the limitation of running in the same environment as Android, hence they are exposed to some attacks [15, 16, 17]. In this work we leverage *ARM TrustZone* to protect our mechanism. As explained in Chapter 2, TrustZone has been adopted to secure a number of services, none similar to T2Droid [93, 91, 94]. For instance, a location-based second-factor authentication for

mobile payments uses TrustZone to protect its secure enrolment scheme [93]. The trusted sensors architecture of [98] uses TrustZone to enable mobile apps to verify the authenticity and the integrity of GPS readings. DroidVault [91] provides a secure data vault on Android devices using the TrustZone. We use ARM TrustZone to protect T2Droid. First, a set of T2Droid components runs inside the secure world, isolated from Android and its apps. Second, a few T2Droid components run in the normal world above Android, but their integrity is verified by components that run in the secure world.

4.6 Summary

This chapter presented T2Droid, a dynamic analysis system protected by the ARM TrustZone extension for detecting malicious Android apps using traces of API functions calls and syscalls. Current anti-malware mechanisms are designed to run in the same execution environment where malware runs, so some malware is able to disable these mechanisms. T2Droid leverages the TrustZone extension to verify the integrity of its software components running in the normal world. The detection itself is performed inside the secure world. T2Droid achieved accuracy and precision rate of 0.98 and 0.99.

Chapter 5

Authenticity Detection for Android Apps

This chapter presents the design and implementation of TruApp, a software authenticity and integrity verification service for mobile apps. This service aims to ensure that an app running in a mobile device (e.g., a smartphone or a tablet) is genuine and was not modified in an unauthorized way by a third party or the user. TruApp is protected from the mobile OS, apps, and malware by running in the secure world. The design explores static watermarking, dynamic watermarking, and measurements (i.e., cryptographic hashes over code). An actual implementation of TruApp does not have to implement all these mechanisms, only a subset of them. The choice of this subset involves tradeoffs that are summarized in Table 5.1 (see Section 5.4.3 on page 109).

Table 5.1: Summary comparison between the three techniques.

Technique	**Protection**	**Detection**	**Delay**
Measurements	best	best (collision resistance)	worst
Static watermarks	best	high	best
Dynamic watermarks	high	worst	average

We assume that TruApp is provided by a *TruApp provider*. Moreover, we assume that an app is provided by an *app vendor*. This vendor may be designated a *service provider*, when the app is an interface for a service, e.g., a home-banking app that provides access to the bank's services. In some cases we use the term *TruApp instance* to clarify that we are talking about TruApp running in a particular mobile device.

To perform verification of an app's authenticity, the app vendor provides a TruApp instance with a certificate called a *verification key* (VK). This VK describes the characteristics of the original app and allows TruApp to verify if the app is genuine. The VK's content depends on the detection mechanism(s) used, e.g., it can be just a hash if only measurements are used. VK is encrypted and signed in order to ensure its authenticity, integrity, and confidentiality.

The *authenticity verification process* works essentially as follows. When the app starts running, it contacts the entity to which it has to prove that it is authentic and obtains a VK and a nonce (for freshness, i.e., to avoid replay). Then, the app calls the TruApp service in the mobile device and passes it VK and the nonce. Next, TruApp verifies the signature of VK, extracts the watermarks and/or the measurements, and checks if they correspond to the information in VK. If they do, TruApp returns a signed certificate containing the nonce. The app finally sends this certificate to the entity in order to verify that the app is authentic. Further details of the VK and authenticity verification process are given in Section 5.2.3.

5.1 Use cases

We believe TruApp is useful in several use cases. Here we briefly present two examples.

The first use case is related to service providers, such as a bank and its home-banking app. The bank is concerned that the app may be compromised and used to steal confidential data about the user and user's bank account. Therefore, the app is distributed via a reliable market and includes logic for invoking TruApp. When the user starts to login via the app, the app's authenticity verification process is executed. If successful, the service provider can trust the app to execute as expected. On the contrary, if the app has been compromised or it is not the original app (e.g., a repackaged version), then the service rejects the attempted access. Note that the authenticity verification must be successful before the user provides any login or other sensitive information via the app.

The second use case is inspired by the PCAS SPD (presented in Chapter 2). The PCAS SPD limits the data it provides to the smartphone, but a malicious app might obtain some useful information. The PCAS app that runs in the smartphone could be protected using TruApp. The app would contact the SPD, receive a nonce and proceed with the authenticity verification process. Notice that, in this case, the resulting authenticity is known to a hardware component connected to the mobile device, rather than to an external service or backend as in the previous use case.

5.2 System architecture and design

This section presents TruApp 's architecture and design.

5.2.1 Threat model and assumptions

We leverage the ARM TrustZone hardware protection to run most of TruApp in the secure world, isolated from the mobile OS. We make the same assumptions about the software running in the normal and secure world, the

TCB, and the API to the secure world as were stated in Section 3.1.2 on page 50. A small part of TruApp runs in the normal world. The rest of TruApp verifies the integrity of the normal world and, specifically, the part of TruApp running there, hence we assume neither malware or attackers can compromise it.

We assume the TruApp service provider has a public-private key pair $(\mathtt{Ku}_{tasp}, \mathtt{Kr}_{tasp})$ for some public-key cryptographic scheme (e.g., RSA), with a key size considered secure (e.g., 3072 bits for RSA [126]). That entity keeps the private key $\mathtt{Kr}_{tasp}$ for itself and installs the public key $\mathtt{Ku}_{tasp}$ in the TruApp service in the mobile devices. Furthermore, the service provider also generates a public-private key pair $(\mathtt{Ku}_{ta}, \mathtt{Kr}_{ta})$ for each TruApp instance running in a mobile device, plus a public key certificate $\mathtt{C}(\mathtt{Ku}_{ta})_{Kr_{tasp}}$ (e.g., in X.509 format [127]). Both keys and the certificate are stored in the TruApp instance in the secure world of the device. Finally, every app vendor has also a public-private key pair $(\mathtt{Ku}_{av}, \mathtt{Kr}_{av})$.

We assume the existence of a collision-resistant hash function [128], e.g., SHA-2 with 512-bit output [126]. We also assume a secure symmetric encryption system (e.g., AES with 256-bit keys [126]) with cipher block chaining (CBC) mode.

5.2.2 Architecture

Figure 5.1 depicts the architecture of TruApp. The mobile OS and the apps it executes, and parts of TruApp (i.e., *syscalls tracer* and *TZ Driver*) are run in the normal world. The *syscalls tracer* is a module of TruApp that intercepts and logs kernel level system calls made by an app running in the normal world. This is only used in conjunction with the dynamic watermarking scheme. *TZ Driver* is a kernel driver that supports cross-world communication between

software in the normal world and TruApp components in the secure world. This driver allocates a shared memory buffer that is used for the software in the normal world to pass selected data to TruApp in the secure world, and for TruApp to return back verification results to the requesting app.

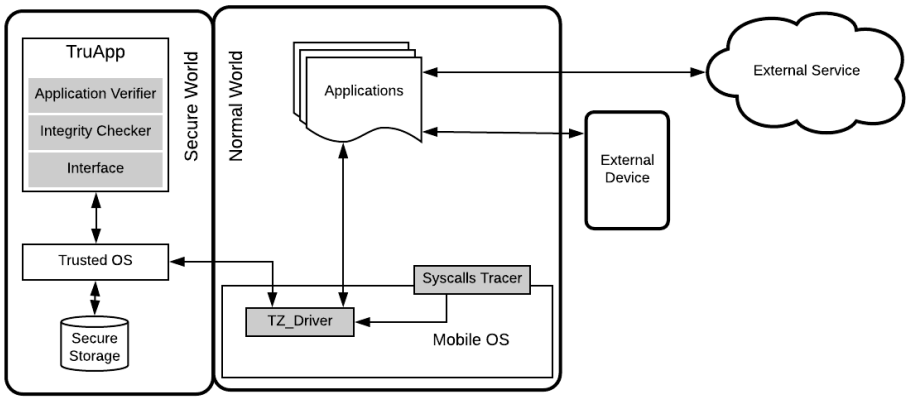

Figure 5.1: Architecture of a mobile device running TruApp. The grey boxes are components of the TruApp service.

The secure world software architecture is designed to run the TruApp modules (*app verifier*, *integrity checker*, and *interface*) on top of a small trusted OS that provides basic OS functions (e.g., process management and file access). The *app verifier* module implements a set of techniques to verify the authenticity and integrity of an app running in the normal world. The *integrity checker* checks the integrity of the TruApp component that runs in the normal world (the *syscalls tracer*) and the Android kernel, as the normal world is vulnerable to attacks. The *interface* module acts as an interface between the normal world and TruApp. It is responsible for receiving and replying to requests, e.g., a request from an app in the normal world for an authenticity verification. It also validates all the incoming data from the normal world and protects against buffer overflows and other input attacks. In addition to the private

memory, a private persistent storage area is reserved in the secure world (shown as *secure storage* in the figure).

5.2.3 Authenticity verification process

This section describes the authenticity verification process, which mainly involves the *app verifier* module and the *interface* module of Figure 5.1.

5.2.3.1 Verification key

The authenticity verification process is based on watermarking and measurements. These techniques require information about the app. For this purpose, the app vendor creates a certificate called a *verification key* (VK) that contains this information (i.e., hash value, static watermark data, and dynamic watermark data).

This VK is *digitally signed*. There are two options for this signature:

1. the signature is issued by the TruApp service provider using its private key $\mathtt{Kr}_{tasp}$, upon request from the app vendor;

2. the signature is issued by the app vendor using its own private key $\mathtt{Kr}_{av}$. As the public key is not in the mobile device, the app vendor provides also a public key certificate $\mathtt{C(Ku}_{av}\mathtt{)}_{Kr_{tasp}}$ signed by the TruApp service provider with its private key $\mathtt{Kr}_{tasp}$.

In the first case, the signature is verified in the TruApp *interface* module using the public key $\mathtt{Ku}_{tasp}$. In the second, the VK comes with the certificate $\mathtt{C(Ku}_{av}\mathtt{)}_{Kr_{tasp}}$, the signature of VK is verified using $\mathtt{Ku}_{av}$ from the certificate, and the signature of certificate is verified using $\mathtt{Ku}_{tasp}$.

VK is *encrypted* using hybrid encryption [129]. This scheme consists essentially in obtaining a random secret key $\mathtt{K}_s$ (e.g., a 256-bit key for AES-256), encrypting the content of VK with $\mathtt{K}_s$, and encrypting $\mathtt{K}_s$ with the public key of the TruApp instance $\mathtt{Ku}_{ta}$. VK consists of three parts: the encrypted content, the encrypted $\mathtt{K}_s$, and the signature.

When VK is received by TruApp in the device, the verification and decryption process consists of verifying the signature as explained above, decrypting $\mathtt{K}_s$, and using this key to decrypt the contents of VK.

This combination of mechanisms ensures the following security properties:

authenticity – VK must have been created by the TruApp service provider or the app vendor, as they are the only entities that have, respectively, $\mathtt{Kr}_{tasp}$ and $\mathtt{Kr}_{tav}$ to sign the message.

integrity – VK cannot be modified and its signature modified to match its content for the same reason as for authenticity;

confidentiality – the contents of VK cannot be disclosed by entities other than the entity who generated the VK or the TruApp instance. As only these entities know the plain text version of the VK or have the private key $\mathtt{Kr}_{ta}$ necessary to decrypt the key $\mathtt{K}_s$ (that is encrypted with $\mathtt{Ku}_{ta}$), and the key $\mathtt{K}_s$ is required to decrypt the content of VK.

5.2.3.2 Overview of the authenticity verification process

It is up to the app to show to the external service or external device that it is authentic (Section 5.1). Therefore, the VK is embedded in the app package that is downloaded and installed from the app market. The app typically gets a nonce – a number that is never reused – from the external service/device,

which it passes together with VK to the *TZ Driver*, which passes them to the *interface* module within the secure world. This process is equivalent to the app calling a remote procedure

```
cert_t TruApp_verify(nonce_t nonce, vk_t vk);
```

where `nonce_t` is the type of the nonce (e.g., a 64-bit unsigned integer), `vk_t` the type of VK, and `cert_t` the type of the certificate returned by the function showing that the verification was successful (otherwise it returns `null`). This certificate is signed using the private key of the TruApp instance ($\texttt{Kr}_{ta}$).

Figure 5.2 presents this process but also shows that between the function call and the result being returned, TruApp interacts with the app. Specifically it obtains measurements and/or watermarks, which it uses to evaluate the authenticity of the app.

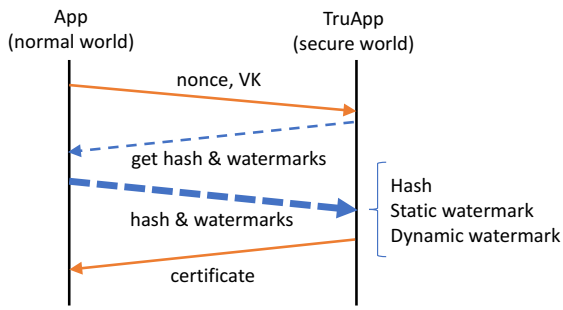

Figure 5.2: Authenticity verification scheme.

The *app verifier* module is composed of three sub-modules, corresponding to the three schemes for verifying authenticity: *measurement checker*, *static watermarker*, and *dynamic watermarker*. Each of these will be described in the following paragraphs.

5.2.3.3 Measurement checker

The *measurement checker* module calculates the hash of the app using a collision-resistant hash function and validates it against the hash value presented in the corresponding VK, in order to verify that the app was not modified. A small change to the app results in a different hash value due to the collision resistance property, so if the new hash value does not match the hash value in the VK, the app (or VK) must have been modified.

This verification is possible because the secure world can access the resources of the normal world. In this mechanism, the secure world inspects the app's APK file stored typically in internal memory (in the part assigned to the normal world).

5.2.3.4 Static watermarker

For static watermarking, we use a scheme in which a watermark is represented by the values of particular bytes in the app bytecode, instead of the usual method of having the value of the watermark stored in a variable (e.g., a string or an integer) of the app source code. The app developer can select which bytes to use and store their positions, i.e., an index to locate a byte, and value of each selected byte in the VK of the app. The greater the number of bytes used and the more scattered they are, the lower the chances of having two different apps with the same values for all of these bytes. The *static watermarker* module running in the secure world reads the position of each byte from the VK and compares its corresponding value with the value of the byte at that position in the target app's bytecode. If any of the bytes do not match, then the app is not authentic.

This checking is possible because the secure world can access all the resources of the normal world. Again, the secure world inspects the app's APK file.

5.2.3.5 Dynamic watermarker

We propose to use *syscall traces* (sequences of syscalls) of the app as dynamic watermarks. As mentioned in Chapter 4, the syscalls made by an app provide relevant information about the runtime behavior of the app. The *syscalls tracer* module running in the normal world intercepts (using the `ptrace` syscall) and logs syscalls made by the target app. When a certain time elapses or a specified number of syscalls has been made (a configuration parameter), the syscalls tracer sends the trace – the sequence of syscalls – to the secure world via *TZ Driver*. We designate this trace $\mathtt{T}_{run}$.

The *dynamic watermarker* module running in the secure world does the verification essentially by comparing $\mathtt{T}_{run}$ with a *reference trace*, obtained with the genuine app, that comes in VK: $\mathtt{T}_{ref}$. This cannot be a trivial comparison of two call sequences because two executions of the same app normally do not produce the same trace, as the exact order of syscalls depends on timing and interaction with other components. To take these effects into consideration, we do a similarity comparison using the Needleman-Wunsch algorithm [130]. This algorithm was designed for finding similarities in the amino-acid sequences of two proteins, not traces. The algorithm essentially compares sequences of letters: it adds points when a match is found, subtracts points when a gap is found (i.e., a match that requires discarding letters in one of the sequences), and subtracts more points when a mismatch is found. The algorithm solves the dynamic programming problem of finding and evaluating the best match between two sequences.

In TruApp, every syscall in $\mathtt{T}_{run}$ and $\mathtt{T}_{ref}$ is converted to a letter using some deterministic criteria. Next, the algorithm configured with appropriate values is used to compute the similarity of the two traces (in the experiments, a match had 4 points, a gap -1, and a mismatch -2). Finally, a threshold $\mathtt{Tresh}_{\neq}$ is used to decide if the apps that produced the traces were the same (result above or equal to $\mathtt{Tresh}_{\neq}$) or not (result below).

5.2.4 Normal world integrity verification

This section presents the *integrity checker* module of Figure 5.1. This module is essentially the *integrity checker* module we presented in Chapter 4. The components of TruApp that run in the normal world, *syscalls* tracer, is vulnerable to malware that infects this environment. Moreover, the behavior of this component may be compromised if the mobile OS is infected. Therefore, it is important to check the integrity of the *syscalls* tracer and the Android kernel, i.e., if they were modified. This process was explained in Chapter 4.

5.3 Implementation

We implemented a prototype of TruApp on an i.MX53 QSB board. The board is configured with the same runtime environment that was implemented for DroidPosture service (explained in Chapter 3).

As shown in Figure 5.1, TruApp has several components, both in the normal and the secure world. The *syscalls tracer* is the same as the one in T2Droid (in Chapter 4) that allows it to trace the syscalls made by a process. This module records only the name of each system call. The trace data is stored in a data structure to be processed and analyzed by the *app verifier* module in the secure world.

In the secure world, we implemented the *app verifier*, *integrity checker*, and *interface* modules. The implementation of *integrity checker* and *interface* are based on the *integrity checker* and *interface* modules that were described in the Chapter 4. Further details of these modules was given in Chapter 4. The *app verifier* module was designed to access the app's APK in the normal world to verify its integrity and authenticity. The implementation of the Needleman-Wunsch algorithm was based on the *seq-align* library [131].

5.4 Experimental evaluation

This section presents the experimental evaluation of TruApp in terms of detection and performance.

5.4.1 Authenticity verification

We conducted experiments to evaluate the detection performance of the three authenticity verification techniques individually (measurements, static watermarks, and dynamic watermarks). These experiments assess the ability of TruApp to play its role and enables app vendors to choose which verification techniques to use. In the experiments, every non-genuine app provided TruApp with the VK of the corresponding genuine app, in an attempt to be considered genuine by TruApp; otherwise these apps could not possibly show to an external service or device that they were genuine.

We started by using TruApp to compare if pairs of apps could be taken to be the same. For that purpose we used a set $\mathcal{A}$ of around 20 random apps downloaded from the Play Store. For every app A in $\mathcal{A}$ with verification key VK_A, we used TruApp to verify if all other apps A' in $\mathcal{A}$ could be taken to be A, by providing VK_A as the verification key. In all cases all three techniques said the app was not genuine, as expected, as the apps were in fact different.

5.4.1.1 Datasets

After that initial phase, we did experiments with apps that were similar, i.e., we compared if a repackaged app A' could be taken for the genuine app A. We used two datasets for that purpose: (1) *manually repackaged apps* and (2) *real repackaged apps*.

For creating dataset (1), we downloaded 41 legitimate Android apps from Google Play Store and repackaged them ourselves. The repackaging involved the following steps: (i) unpack the APK file using the Apktool [132]; (ii) convert the bytecode (DEX file) to Smali code (human-readable bytecode) using the same tool; (iii) add to the Smali code a simple malicious code snippet that deletes the user's contacts (taken from [133]); (iv) modify the file `manifest.xml` to give the app more permissions (in this case to read and write the contacts) and to trigger the code when the mobile device finishes booting; (v) repack the app with the Apktool; and (vi) sign the APK file and add a self-signed certificate.

For dataset (2), we had first to find repackaged apps and the corresponding genuine apps (from Play Store), which was not simple. We found 20 pairs of apps that met these conditions (see Table 5.2). The repackaged apps were mostly *mod games*, i.e., games modified to give the player some kind of advantage (e.g., unlimited gems in Clash of Clans).

5.4.1.2 Verification

Both *measurements* and *static watermarks* always detected that the repackaged apps were not genuine. In the case of measurements, this happened because repackaging modifies the app bytecode, leading to a different hash value. In the case of static watermarks, the cause was the fact that the repackaging

Table 5.2: Detection rate for dataset (2), real repackaged apps.

APKs \ $\texttt{Tresh}_{\neq}$	**1700**	**1750**	**1800**	**1850**	**1900**
Angry Birds 7.5	0	0	0	0	0
Bomb Squad Pro 1.4.121	0.95	1	1	1	1
CCLeaner 1.19.76	0	0	0	0	0
Clash of Clans 9.24.15	0	0	0.05	0.05	0.05
Clash Royale 1.9.2	0	0	0	0	0
Crossy Road 2.4.4	0	0	0	0	0
FIFA Mobile Soccer 6.1.1	0.1	0.1	0.15	0.2	0.2
Flags Quiz 2.4	1	1	1	1	1
Flick Kick FootballLegends 1.9.85	0	0	0	0	0
Last Day on Earth 1.5.6	0	0	0	0	0
Last Hope TD 3.31	0.25	0.25	0.25	0.25	0.25
Magikarp Jump 1.1.0	0	0	0.1	0.1	0.1
Mo n Ki World Dash 1.6	0.15	0.15	0.2	0.2	0.35
Once Upon a Tower 3	1	1	1	1	1
Realm Defense 1.8.4	0.05	0.05	0.1	0.1	0.1
Sniper 3D Assassin 2.0.2	0	0	0	0	0
Super Mario Run 2.1.1	0	0	0	0	0.1
Zombie Castaway 2.8.1	0	0	0	0	0.05
8 Ball Pool 3.10.3	0	0	0	0	0.1
8 Ball Pool 3.10.1	0	0	0	0	0

introduces the code snippet and creates a shift of the bytes past the position where the snippet is added, so the original byte values are not found in the expected position. Therefore, we concluded that for evaluating verification the important case is dynamic watermarks.

To measure the detection performance of *dynamic watermarks*, we need to obtain execution traces, which involves executing all apps with the same sequence of inputs. For this purpose, we run and traced the apps using the Monkey app exerciser [123] to generate a stream of user events such as clicks, key presses, and screen touches. If Monkey is re-run with the same seed value,

it can generate the same sequence of events. Due to the complexity of using Monkey and executing these experiments in the board, we run the apps in Google's Android emulator [124], setting it to emulate an ARM CPU. We run and collected 30 traces for each of the manually repackaged apps and 20 traces for each of the real apps.

We considered different threshold values for the Needleman-Wunsch algorithm to decide whether the traces are the same or not. For each threshold value, we measured the *detection rate*, which is the ratio of non-genuine apps that are detected to be so. For dataset (1), the detection rate was 0, i.e., the dynamic watermarking technique failed to detect non-genuine apps. The reason for this is that the malicious code snippet runs only after a reboot, so it was never executed and the repackaged apps were confused with the genuine ones. We did the same experiment for dataset (2). The results of these experiments are shown in the Table 5.2. Again, the results were quite bad, with values near 0 except for 3 apps: Bomb Squad Pro 1.4.121, Flags Quiz 2.4, and Once Upon a Tower 3. The reason for this is that most of the repackaged apps have a behavior that is very similar to the genuine app.

For dataset (1), we also compared the 30 traces of each app with the reference trace of each of the other apps. We measured six common performance metrics (defined in Chapter 4 Section 4.3 on page 82) for different threshold values. Consider that: a positive (P) is an app detected by TruApp to be non-genuine; a negative (N) is an app classified as genuine; a wrong positive or negative are denominated respectively false positive (FP) and false negative (FN); the opposite are a true positive (TP) and a true negative (TN).

The results of these experiments are shown in Table 5.3. The best performance depends on the metric considered, e.g., it was for $\texttt{Tresh}_{\neq} = 1830$ for the

Fmeasure metric. Notice that the accuracy does not vary much with $\texttt{Tresh}_{\neq}$, wereas recall goes down and precision up.

Table 5.3: Evaluation of dynamic watermarking.

$\texttt{Tresh}_{\neq}$	**Accuracy**	**FPR**	**FNR**	**Recall**	**Precision**	**Fmeasure**
1750	0.977	0.019	0.051	0.949	0.894	0.921
1770	0.975	0.017	0.061	0.939	0.902	0.920
1790	0.977	0.015	0.071	0.929	0.910	0.919
1810	0.978	0.012	0.083	0.918	0.928	0.913
1830	0.980	0.009	0.092	0.908	0.947	0.927
1850	0.981	0.003	0.112	0.888	0.978	0.930

Table 5.4 summarizes the values of the metrics for the three techniques. For dynamic watermarks we consider the values of Tables 5.2 (the worse case; some metrics are not filled-in as there is no value for TN) and Table 5.3 (for $\texttt{Tresh}_{\neq} = 1830$). The table shows essentially that measurements and static watermarks provide better detection than dynamic watermarks.

Table 5.4: Comparison of the three techniques.

Technique	**Accur.**	**FPR**	**FNR**	**Recall**	**Prec.**	**Fmeas.**
Measurements	1	0	0	1	1	1
Static watermarks	1	0	0	1	1	1
Dynamic watermarks (Table 5.2)	–	–	1	0	–	–
Dynamic watermarks (Table 5.3)	.980	.009	.092	.908	.947	.927

5.4.2 Performance overhead

We also evaluated the performance overhead incurred by the authenticity verification techniques to study which technique is best in this aspect.

In order to evaluate the overhead of the measurements technique, we evaluated the time for the *measurement checker* module to calculate the hash of the

app using SHA-512 and compare it against the hash value present in VK. We repeated this experiment for different sizes of APK files. The results in Table 5.5 show that the time (t) in seconds grows linearly with the file size ($file\_size$) in bytes. The trend observed is $t = 0.9388 milliseconds/byte \times file\_size - 0.0562$.

Table 5.5: Time to do measurements.

Size (MBytes)	**Time (ms)**
3.3	3,090
4.9	4,580
13.3	12,430
17.5	16,350
18.6	17,370
25.6	23,970
28.3	26,510
37.7	35,160
59.0	55,540
91.5	85,790

We also evaluated the overhead of static watermarking. The *static watermarker* module running in the secure world reads the position of each byte listed in the *VK* and compares its corresponding value against the value of byte in that position in the target app bytecode. Therefore, we measured the time (t) to perform these operations. Since this time depends on the number of bytes (n) used as watermark, we repeated this experiment for different numbers of bytes (see Table 5.6). The trend is $t = 3.0056 milliseconds/byte \times n - 90.24$.

For dynamic watermarks, we did not measure the time of the whole detection process as the time to extract the traces is configurable. We measured the total time required for the *syscall tracer* module in the normal world to transfer the trace data (syscalls traces) to the *interface module* in the secure world. This time includes the performance delay introduced by context switching between the two worlds, copying the trace data into the shared buffer, and sending it

Table 5.6: Time to do static watermarking.

No. Bytes	Time (ms)
4	66
8	68
16	72
32	81
64	97
128	130
256	196
512	1,556
1024	3,059
2048	6,071

to the secure world. For this, we measured the time for copying different sized chunks of data into the shared buffer and sending them into the secure world. The average throughput to perform the above operations is 17.51 MB/s. We also measured the time for the dynamic watermarker to convert a syscall trace into a sequence of an alphabetical letter and to compare the traces to detect if they are same or not. We repeated this experiment for different numbers of letters (l) in the converted sequence of letters (200. 400, 600, 800, and 1000 letters in length). The results are shown in Table 5.7. For each number of events (with each letter corresponding to one event), the total time (t) to complete the above operations is shown in the last column of the table. The trend is $t = 0.5042 milliseconds/letter \times l + 58.534$.

Table 5.7: Time to do trace conversion and comparison.

No. Letters	Conversion (ms)	Comparison (ms)	Total (ms)
200	96.97	107.43	204.4
400	114.68	108.72	223.4
600	132.53	188.42	321.0
800	159.73	312.63	472.4
1000	168.89	415.21	584.1

Finally, we evaluated the performance overhead incurred by the normal world integrity verification process. The *system verifier* checks the integrity of the normal world by calculating hashes of the bootloader, Android kernel, init, and app_process using SHA-512, and comparing them against their known-good values. The *tracer checker* does the same operations for the *syscall tracer* module running in the normal world to verify its integrity. Therefore, we measured the times to perform those operations. The results are the same as those shown in Chapter 4 Table 4.3 except for the *API call tracer*.

5.4.3 Tradeoffs of the different detection techniques

Table 5.1 on page 91 presents a comparison of the three techniques. In relation to protection from the normal world (second column), it is now clear that the measurements and static watermarks techniques are executed only in the secure world, so they have the best protection from the normal world. In contrast, the dynamic watermarks technique requires running a module in the normal world (*syscalls tracer*), so its degree of protection is lower, although still high due to the use of the integrity verification mechanisms.

In relation to their ability to detect if an app is not authentic (third column), measurements are clearly the best technique because they leverage the collision resistance property of hash functions. For the other two, their capacity to detect if an app is not authentic depends on the modifications made to the app, modifying, respectively, the bytes checked with the static watermarks or the syscall sequence for the dynamic watermarks. Our experimental results show that this is high for static watermarks, but lower for dynamic watermarks.

In relation to the time to execute the technique (third column), the highest overhead occurs when computing cryptographic hashes, so measurements tend

to be the worst in overhead, followed by dynamic watermarks. In contrast, static watermarks are lightweight.

This discussion and Table 5.1 show clearly that there are tradeoffs that the designer of a TruApp implementation can explore, allowing optimization of different metrics for the specific use case.

5.5 Related work

In order to mitigate app repackaging in the context of mobile devices, Jang et al. [134] proposed a static watermarking mechanism based on steganography techniques. It embeds watermarks into Android apps by reordering the sequence of instruction in the basic blocks of the app DEX files. AppMark [135] and Droidmarking [136] are examples of dynamic watermarking mechanisms that embed a watermark generator into Android apps that are guaranteed to be executed and dynamically create a watermark instance at runtime. The watermark code is combined with the original app with strong data dependency in such a way that it is difficult for attackers to identify and modify the code by analyzing data dependency. AppInk [137] has adopted traditional graph-based dynamic watermarking, implementing it on Android apps. AppInk takes as input an app's source code and a watermark data. However, adversaries can remove the watermark code embedded in the target app using reverse engineering tools or disable the protection implemented external to the app (e.g., in the mobile OS). In contrast, TruApp runs the protection or the watermark code inside the secure world, isolated from the mobile OS, apps and malware by leveraging the TrustZone.

ARM TrustZone has been used to implement watermarking [138, 139]. TrustICE [138] uses TrustZone-based watermarking to dynamically protect memory regions in the normal world. The other work uses watermarking for video, rather

than software. We use ARM TrustZone in a very different way: for assessing the authenticity and integrity of apps and to ensure that the assessment mechanisms are isolated from Android and its apps.

5.6 Summary

This chapter presented TruApp, a software authenticity and integrity verification service for mobile devices. TruApp is protected using ARM TrustZone. It is executed mostly in the secure world, but also verifies the integrity of those parts of the normal and of the TruApp code that run in the normal world. TruApp uses watermarking and measurements to assess the authenticity of mobile apps. We presented an implementation of TruApp for the i.MX53 QSB and an experimental evaluation of this implementation.

Chapter 6

Full Recovery and Protection from Ransomware

This chapter describes the design and implementation of ransomSafeDroid, a TrustZone-based backup and recovery service for protecting mobile devices from strong attacks that compromise the filesystem (e.g., encrypting everything as in the case of crypto-ransomware) and that make the device (the normal world) partially or fully unusable. We leverage TrustZone to protect ransomSafeDroid from malware running in the normal world. Because ransomSafeDroid runs in the secure world, it is secure despite the normal world, including the mobile OS, being compromised. ransomSafeDroid is able to make a periodical backup of files in the normal world to a local storage partition in the secure world, and to push these backups to external storage (e.g, a cloud service). It offers fast incremental backups which capture only the changes made to files since the last backup and efficient storage compression and deduplication to remove redundant data, thus saving backup storage space.

To the best of our knowledge, in the existing TrustZone literature it is always the normal world that initiates a call to a trusted service in the secure world,

using the `SMC` instruction [33, 98, 94, 35, 36]. However, such a mechanism might be blocked by the ransomware, hence we have to follow a different approach and make the secure world active, instead of passive. Specifically, ransomSafeDroid is scheduled to run automatically at specific times, without the intervention of the normal world. For this to happen, a hardware timer is used to generate a periodic interrupt which forces the execution of ransomSafeDroid in the secure world. This means that ransomware running in the normal world cannot stop it.

6.1 Android storage architecture

Mobile devices such as smartphones normally have three types of read/write memory: volatile RAM, non-volatile solid-state *internal storage* (e.g., eMMC), and an optional non-volatile solid-state *SD card*. Android apps can store persistent configuration files and data in the device's internal storage and in the SD card. Android uses different filesystem partitions to organize files and folders on the storage device. There are typically six partitions for the internal memory plus one partition on the SD card, i.e., bootloader (`/boot`), kernel (`/system`), device recovery (`/recovery`), user data (`/data`), data/component cache (`/cache`), miscellaneous settings (`/misc`), and SD card (`/sdcard`).

Android used the YAFFS2 [140] filesystem for the various internal partitions including `/system` and `/data`. YAFFS2 is lightweight and optimized for NAND internal storage. Recently, Android transitioned to *ext4* as the default file system for these partitions [141]. For compatibility, Android also provides a filesystem to access a FAT32 formatted external storage, as this is a commonly used file system on SD cards.

6.2 File backup schemes

Backup refers to copying files to an alternative storage location, typically secondary or remote storage, in order to prevent data loss due to human action, hardware and software failures, lost/theft of the device, or accidents/disasters. There are three main backup techniques: full backup, incremental backup, and differential backup.

In a *full backup*, all the files that have been selected for backup are copied. This approach is efficient when the number and aggregate size of files is small, but otherwise it can be slow. Moreover, it can be expensive if performed over a cellular network when there is a per byte tariff or a cap on data volume.

Incremental and differential backups aim to solve the issues of delay and cost. Both require an initial full backup, then only those files that have been modified or newly created are copied when subsequent backups occur. The difference between these two is that an *incremental backup* considers all changes since the previous backup (full or incremental), while a *differential backup* includes all files that have been changed or added since the last full backup. These two schemes result in faster and smaller backups in comparison with full backup, as long as only a small percentage of files changes before each subsequent backup. The advantage of reducing the size of backups is not only less storage space is needed, but also less data needs to be transferred over the network in the case of remote backups.

All of the above backup methods can be combined with other methods of reducing the backup storage space and network bandwidth, such as data compression [142] and deduplication [143, 144].

There is a vast range of *compression* techniques, but all are based on the idea of removing redundancy in order to store data in a more compact form. Compression techniques may be lossless (the original data can be fully recovered) or lossy (the original data cannot be entirely recovered). For backup purposes, lossless compression is almost always used.

Deduplication aims to remove redundancy with respect to files stored in the same data store. In file-level deduplication, a file is not copied if it already exists in the store. In block-level deduplication, files are broken in blocks of the same size; a block is not copied if it already exists in the store.

6.3 ransomSafeDroid

This section presents the architecture and design of ransomSafeDroid.

6.3.1 Threat model and assumptions

We assume a device with an ARM processor with the TrustZone extension. ransomSafeDroid runs in the secure world, isolated from the normal world. We make the same assumptions about the software running in the normal and secure world, the TCB, and the API to the secure world as were stated in Section 3.1.2 on page 50.

The device is configured with a hardware timer reserved for the secure world. While malware or attackers might be interested in disabling this timer, we assume they cannot access or compromise the resources assigned to the secure world. We also assume that, if the user wishes to initiate a restore of their files, the user interface for this operation is secured by using TrustZone (this has previously been addressed in [36]). In all cases we assume that the attacker does not have physical access to the device.

6.3.2 Architecture

The architecture of ransomSafeDroid is shown in Figure 6.1. The normal world runs a mobile OS and apps, whereas the secure world runs ransomSafeDroid on top of a small trusted OS that provides basic OS functions (e.g., process management, file access, and memory management).

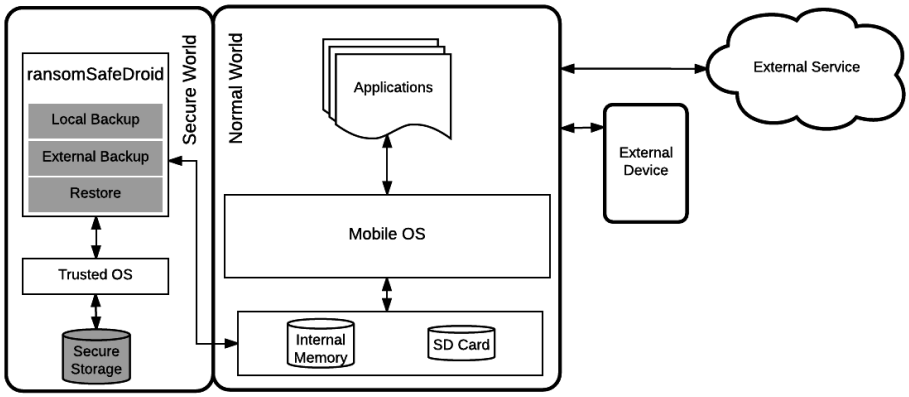

Figure 6.1: Architecture of a mobile device running ransomSafeDroid. The grey boxes are components of ransomSafeDroid.

The *secure storage* is a private persistent partition used for local backup storage. It is isolated in the secure world, i.e., it cannot be accessed by the normal world. This way, backups are protected from malware running in the normal world despite the mobile OS being compromised by ransomware.

ransomSafeDroid has three software modules: local backup, external backup, and restore. The *local backup* module copies files in the normal world into the local secure storage, whereas the *external backup* module pushes backups stored in secure storage to an external device, a remote server, or cloud computing service (e.g., Amazon S3 or Google Drive). The *restore* module is responsible for restoring files and system components (e.g., kernel,

init, etc.) in the normal world from local or remote backups, or recovering the full system if the device is no longer usable, i.e., removing the effects of ransomware attacks or other system failures.

6.3.3 Active secure world

To protect mobile devices from ransomware, not only must the backups be protected, but the adversary must be prevented from denying the execution of the backup/restore operations. Therefore, backups/restores cannot be initiated by a process in the normal world. This implies that the secure world has to be active, i.e., has to run RANSOMSAFEDROID without being called. As noted earlier as far as we are aware, this is the first such TrustZone use proposed in the literature.

In order to generate an interrupt to initiate backup/restore operations, RANSOMSAFEDROID configures a hardware timer to be accessible only by the secure world, preventing the normal world from disabling it. This timer will generate a periodic interrupt that triggers the execution of the secure world. In the discussion that follows we focus specifically on the local backup module as this is likely to be the most frequent operation.

The period T_{sw} (secure world execution period) is configurable. There is a tradeoff when setting T_{sw} as with a longer period the higher the probability some important file may not be backed up when an attack happens; while a shorter period increases the overhead, as the normal world execution is more frequently interrupted and the more likely it is that there are no changes in files to be processed.

The assigned hardware timer is not part of the ARM processor itself, but of the device, so it depends on the specific hardware used. Section 6.4 explains the configuration for the i.MX53 board that was used to implement the prototype.

Next we present the main components of RANSOMSAFEDROID (as were shown in Figure 6.1).

6.3.4 Local backup module

The local backup module copies files from the normal world and saves them to the secure storage in the secure world, for a later restore if the device is attacked and files are encrypted by ransomware.

This module creates in the secure storage (in the secure world) an index of the files and directories in the filesystem that are being backup (from the normal world). This index stores a list of files and their attributes, along with hashes over the files. This data is used to track the files in the index that have been updated since the last backup.

This module utilizes an incremental backup to save both space and time. Files are divided into blocks of data and each block is separately indexed by its hash/checksum. The scheme uses rolling checksums based on the rolling checksum algorithm from the *rsync* tool [145] to compare the file blocks, for determining the difference between two files and to save only the difference. This scheme allows identification of even small differences in large files, hence minimizing the number of blocks that have to be stored in the backup.

As explained in Section 6.2, the first backup has to be a full backup. Doing a full backup of an Android filesystem takes some time, as it typically involves copying gigabytes of data. Moreover, mobile devices tend to be slower than PCs, with respect to processor speed, internal bandwidth, and memory speed. However, the overhead of the full backup does not have an impact on usability, as this full backup should be done before the device enters normal operation. For example, it can be done by the device manufacturer before it is sold to the end-user.

Moreover, the backup module uses both compression and deduplication. In terms of compression, the idea is simply to compress the files using zip-like lossless compression algorithms. In relation to deduplication, the idea is to use hashes of the file blocks to identify and remove redundant data across files and generations of backups. If the backup repository already contains a particular block of data, it will be re-used and the duplicated block of data will be replaced with a link. This feature can significantly reduce the amount of storage space needed for backups, which is important for mobile devices since they normally have limited storage space.

The secure storage should be large enough to contain the full backup and the increments, but it cannot have infinite space. To deal with this limitation, the major mechanism is to make use of an external backup (see next section). Moreover, if the space available goes below a certain threshold, the user may be told explicitly to do an external backup, e.g., using an Android notification (that appears on the top of the screen).

6.3.5 External backup module

Local backups are effective against ransomware, but not against other more classical threats, such as the device being lost, stolen, or destroyed. Therefore, we extend the basic RANSOMSAFEDROID design with the ability to perform backups to external storage, e.g., to remote servers or to a remote cloud computing storage service.

As local backups are already compact, the remote servers simply maintain a copy of the local backups. A naive solution would be to simply copy all new and modified files to the remote storage whenever they appear in the (local) secure storage. However, this would considerably increase the overhead of doing the backups. Therefore, we use an opportunistic and cautionary

approach: whenever a local backup module is executed, it measures the time it takes to run, i.e., t_{lb} (local backup time). Then, it passes control to the external backup module that copies files to remote storage during at most $t_{max} - t_{lb}$ units of time (if positive). The interval of time t_{max} (maximum time) is a configurable parameter that indicates the maximum time the backup process should take, i.e., that the secure world should run in every period T_{sw}. It is important to limit this time because ARM CPUs do not do time sharing between the two worlds, hence when the secure world is running the normal world is blocked (and vice-versa). These times are represented in Figure 6.2.

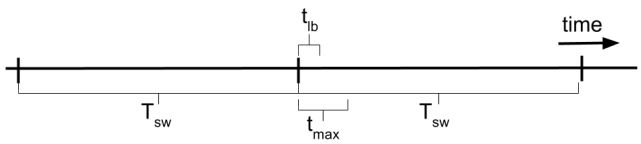

Figure 6.2: Backup period and other relevant times.

External backups can be done in two ways. The first is by connecting a storage device (e.g., an external disk or an SD card) directly to the mobile device. This solution requires allocating the I/O devices needed for this purpose to the secure world in order to prevent malware in the normal world from corrupting such backups. Second, the backup can be done via a network. As we explained earlier, the secure world does not contain a network stack in order to reduce the size of the TCB. Therefore, backups via the network have to be done with the assistance of a gateway app in the normal world (not shown in the figure, as this is optional). Additionally, end-to-end encryption and message authentication codes for security must be used (similarly to the solution in [94]). As a result, malware in the normal world may be able to prevent this communication, but cannot corrupt it. Nevertheless, recall that the purpose of remote backups is not to protect against ransomware but rather to protect against other forms of threats.

6.3.6 Restore module

This module has the role of restoring files that are encrypted or corrupted when the normal world is still usable/working. It has the ability to restore files/backups from local or remote storage.

The restore module is also responsible for performing recovery of the full system when the device is no longer usable. It is capable of installing/repairing/upgrading all the system software running in the normal world, including the mobile OS kernel. For this purpose, copies of system components when they are in a clean state are stored in a persistent partition in the secure world.

The restore operation of mobile devices (e.g., smartphones) is normally started during the boot process. With devices that have TrustZone, we assume that the secure world starts to run first before the normal world, so the secure world kernel starts the restore module when the device boots. The restore module displays a limited user interface that provides users options, such as to selectively restore files or system components and to recover the full system. In order to start the restore module, users can use a combination of special key presses during the early stage of the boot process.

When the system is not fully compromised, the restoring of files can be also started/initiated using a *restore app* that is executed in the normal world if the normal world is and allows the user to define how the restore is done. This app calls the restore module in the secure world using the `smc` instruction. The authenticity of the restore app may be checked using TruApp (explained in Chapter 5).

Before any restore operation, the user has to provide authentication credentials in order to prove to RANSOMSAFEDROID that he or she is a legitimate user, rather than an adversary.

When a restore is done after a ransomware attack, most likely the most recent backups will consist of encrypted files. Incremental backups allow dealing with this problem by not recovering the increments that correspond to encrypted files. The detection of which increments correspond to these encrypted files can be done using a few heuristics. First, when a whole filesystem is encrypted by ransomware, there must be an easy way to observe peak in the number of files to be backup, from a few to many. Second, the restore module may include a scheme to detect which files are encrypted, e.g., using entropy analysis [146]. Moreover, file signatures can be matched to the extension of the file name to see if the contents are of the expected type. However, should either of these be insufficient then the user will have to identify the date to which the files should be restored (i.e., the versions of files before this date will be restored).

The restore operation tends to be quite slow, because it has first to copy the initial full backup, then all the changes that were made. However, while the backups are executed during normal operation, restore is an exceptional operation, executed only when a problem occurs (e.g., a ransomware attack). Therefore, the time to restore is not considered critical for the usability of ransomSafeDroid.

6.4 Implementation

We implemented a prototype of ransomSafeDroid on an i.MX53 QSB board based on the runtime environment setup presented in Chapter 3. This environment provides runtime support to implement ransomSafeDroid modules running in the secure world. In the current prototype, we allocated 50 GB of secure storage space from a total of 128 GB space available on a microSDHC UHS-I Class U3 card. This card supports speeds up to 90 MB/s for reads and 80 MB/s for writes.

In the prototype, the local backup module is based on *bup* [147], an open source backup tool, selected after comparing the performance of five similar software packages. *bup* is an efficient file backup software based on the *git* packfile format. *bup* offers incremental backups, compression using the *zlib* library (the default in the *git* packfile format), and storage deduplication, thus providing backup time and space savings. This tool has several modules, including the *index* module to create an index of files, and the *save* module that creates a new backup. Unfortunately, *bup* is written in Python, which requires adding the Python runtime to the TCB. Nevertheless, this is just a prototype and for production purposes *bup* should be replaced by code written in C/C++ or another language that does not require a runtime engine. Moreover, the *bup* website acknowledges that "Writing more parts in C might help with the speed" [147].

In order to make the secure world active, we used the *enhanced periodic interrupt timer* (EPIT) available on the i.MX53 board [148]. EPIT is a 32-bit set-and-forget timer. A driver for the EPIT timer has been implemented by the Genode project in the *base-hw* kernel running in the secure world, allowing the configurable periodic execution of the secure world. In i.MX53, groups of I/O devices are assigned to one of the worlds using configuration bits of the *central security unit* (CSU), similar to what happens with the protection controller used on ARM's versatile express platform [110]. In our configuration, a group containing the EPIT is assigned to the secure world. This is done by modifying a Genode configuration file (`csu.h`), where hardware assigned to the secure world is tagged `secure`, whereas hardware assigned to the normal world is tagged `unsecure`.

We focused most of our implementation effort on the mechanisms just described, which are the most relevant for the normal operation of RANSOMSAFEDROID.

In contrast, the implementations of the external backup module, the restore module, and the restore app are not finished yet.

6.5 Experimental evaluation

In this section, we evaluate RANSOMSAFEDROID in terms of backup latency and storage efficiency by considering three different circumstances: initial backup, runtime backups, and null backups. We consider only local backups, because this is the main functionality of RANSOMSAFEDROID and the performance of remote backups depends strongly on the t_{max} and t_{lb} parameters as well as the properties of the communication with the remote store. As explained in Section 6.3.6, we do not evaluate the restore time because this is considered to be an exceptional operation that does not impact usability.

6.5.1 Initial backup

We conducted experiments to understand the cost of an initial (full) backup in terms of time and storage space. To make these experiments realistic, we used a real Android filesystem with a 1.1 GB storage size. We ran RANSOMSAFEDROID to perform a full backup of all files in this filesystem, and measured elapsed time and size of the backup created in the secure storage (in the secure world). Additionally, we repeated this experiment for different filesystem sizes: 2.2 GB, 3.3 GB, 4.4 GB, and 5.5 GB. For this purpose, we prepared different filesystems by adding new files to the real Android filesystem in multiples of 1.1 GB. We use the *genbackupdata* tool [149], which generates test data sets for performance evaluation of backup tools. Each data set consists of files and directories. Files can be either text files or binary files. In these tests we used 10 equally sized files and set equal percentages of text files and binary files. The results of these experiments are shown in Figure 6.3.

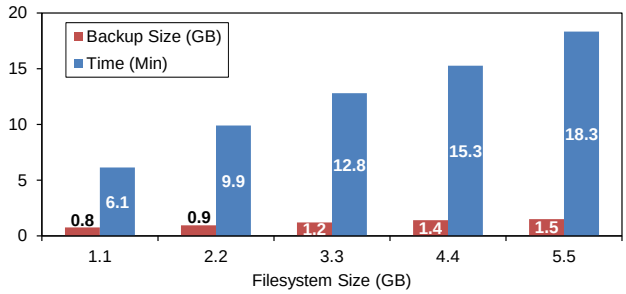

Figure 6.3: Initial/full backup with different Android filesystem sizes.

A first conclusion is that the full backup takes a considerable time, e.g., 6.1 minutes for the smaller filesystem and 18.3 for the largest. This may seem to be a deterrent for the use of this service; but, as explained in Section 6.3.4, the full backup can be done before the device is in normal use, even before it is sold to the end-user. A second conclusion is that the time grows approximately linearly with the size of the filesystem. Finally, it is noteworthy that the read/write speeds of the SD card are not the performance bottleneck, as the time to read and write 5.5 GB from the card at the (maximum) speeds of 90 and 80 MB/s is 2.16 minutes, which is far less than the observed 18.3 minutes. The bottleneck is the CPU speed, as the CPU used on the i.MX53 has 1 GHz clock speed.

Next, we did an experiment to show that it is possible to reduce the time of the full backup by doing it in steps. For that purpose, we evaluated the performance of a full backup of each partition or sub-tree in the filesystem. The 1.1 GB Android filesystem in the current prototype has 5 partitions: `cache`, `recovery`, `system`, `data`, and `sdcard`. For each partition, we measured the total time to complete a full backup of the partition and the storage space used by the backup data. We show these results in Figure 6.4. It is possible to observe that the total time for the full backup is greater than the 6.1 minutes

observed in Figure 6.3. However, the times to backup the individual partitions is smaller, so they might be easier to do during idle times, if that was required.

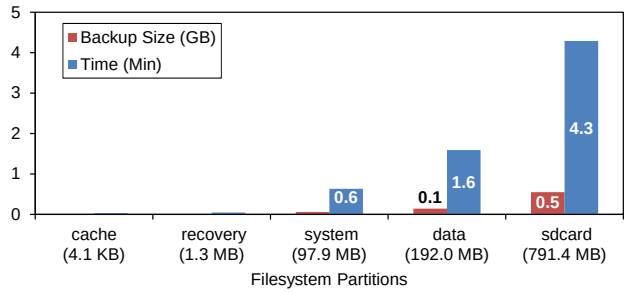

Figure 6.4: Initial/full backup of Android filesystem partitions (1.1 GB filesystem).

6.5.2 Runtime backups

The overhead of runtime backups depends on two factors: the period of the incremental backups (T_{sw}) and their size. As the period is configurable, we did not take it into account in the evaluation. However, one might assume that this could occur when the device is otherwise unused and recharging (perhaps each night).

We considered the case of the size of a filesystem increasing by 1% in every step to emulate incremental changes. For this purpose, we took as a basis the full backup of the 1.1 GB filesystem, then we created new files using the *genbackupdata* tool as described earlier.

We performed two experiments by considering the number of new files created in each step after the initial full backup of the Android filesystem. In the first experiment, we increased the size of the filesystem by 1% with 10 equally sized files in each incremental backup; in the second experiment we also incremented the size of the filesystem with different numbers of equally sized files in powers of 10 (e.g., 10, 100, 1000, and 10000 files) in each step. Note that the total

number of bytes that are backed up in each incremental backup are the same in both experiments: 1% of the size of the filesystem. We measured the latency and backup size for the two experiments. We show the results in Figures 6.5 and 6.6.

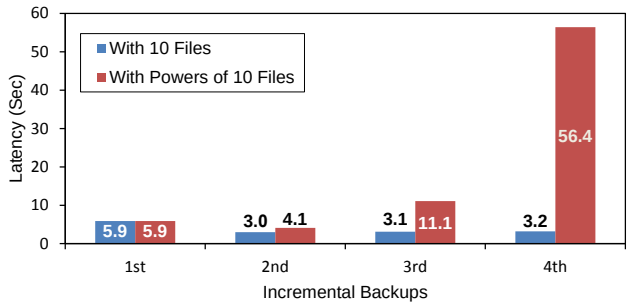

Figure 6.5: Latency to make incremental backups when the filesystem is increased by 1% with 10 equally sized files and with powers of 10 files in each backup.

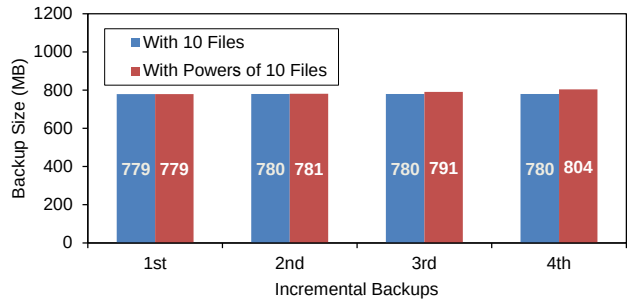

Figure 6.6: Storage space usage to make incremental backups when the filesystem is increased by 1% with 10 equally sized files and with powers of 10 files in each backup.

A first conclusion is that the times are much smaller than those observed for the full backup, as expected as the amount of the data is much smaller (times are in seconds, no longer in minutes). A second conclusion is that backups of more files take more time, e.g., the backup of 11 MB with 10 files and 100 files – second backup – takes respectively 3.0 and 4.1 seconds.

6.5.3 Null backups

Finally, we also measured the time it takes to run RANSOMSAFEDROID and do an incremental backup when the filesystem did not change, i.e., the time spent for a null backup. This may happen quite often, depending on the periodicity with which RANSOMSAFEDROID is executed (i.e., depending on T_{sw}), hence it is important to understand how much time null backups take.

As before, we considered different sized filesystems (i.e., 1.1 GB, 2.2 GB, 3.3 GB, 4.4 GB, and 5.5 GB) and configured RANSOMSAFEDROID to periodically perform 10 null incremental backups for each filesystem size. We show the average of these 10 results for each different size of filesystem in Figure 6.7.

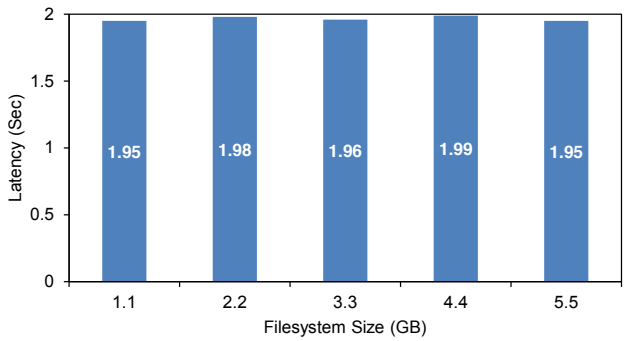

Figure 6.7: Latency of null backups, i.e., incremental backups when the filesystem was not changed.

We observe that the time needed to perform a null backup for all filesystem sizes is approximately 2 seconds on our board. This is the time necessary for RANSOMSAFEDROID to check if any change has been made since the last backup. The time may be even longer if the numbers and size of files in the filesystem are higher. Interestingly, the time seems to be independent of the size of the filesystem analyzed.

We studied if it is possible to reduce this delay of 2 seconds by implementing an alternative mechanism to check if files were modified. For that purpose, we implemented a small script using the *find* command. We obtained a delay of 20 milliseconds, with a standard deviation of 6 milliseconds. This supports our claim that it is possible to improve performance with a more efficient implementation of the backup code, e.g., written in C/C++.

Notice that these times are measured between the moment the secure world starts to run until it returns, so they do not include the time for context switching between the two worlds. It is not possible to measure this latter time using the interrupt caused by the timer, so we created a routine to trigger the secure world by calling the `smc` instruction in the normal world, then returning immediately. This time is measured in the normal world just before and after the trigger. We obtained an average time of 45 microseconds (averaged over 1000 repetitions).

6.6 Related work

There is a large variety of backup software for keeping copies of all data (photos, videos, music files, and contacts) and apps installed on smartphones and tablets. Some backup apps, e.g., *Samsung Smart Switch* [29] and *LG Bridge* [30], transfer data and apps to a PC using a USB cable and vice-versa. Mobile OSes such as Android and iOS also provide utilities that facilitate backing up of data to a remote storage cloud. Furthermore, Android also provides the *Auto Backup API* for developers to add backup functionality to their apps [150]. This API allows apps to backup data by uploading it to the user's Google Drive account, where it is protected by the user's Google account credentials. Nevertheless, all these backup schemes are exposed to malware as they run inside an untrusted execution environment where other apps and the

mobile OS run [15, 16, 17]. Their security is based on a set of assumptions that are often broken: no permissions should be granted to download apps, no vulnerabilities should exist in certain apps, no vulnerabilities exist in the mobile OS, etc. In contrast, we use ARM TrustZone to protect our backup mechanism, based on the isolation it provides.

Several works on secure storage have proposed mechanisms to protect sensitive data such as private keys, that are only accessible to small security critical programs using the TrustZone [35, 36]. However, unlike our work, none of these works provides a backup solution or focuses on protection from ransomware. Moreover, all of these programs are designed such that it is the normal world that calls the secure world. In contrast, ransomSafeDroid leverages an active secure world by using a hardware timer to initiate the secure world without intervention of the normal world.

Currently, most backup apps place the backed up files in rented remote servers or cloud storage, which do not belong to the owner of the files. This may lead to personal or confidential data being accessed without the consent of the owner [151, 152]. However, ransomSafeDroid use a secure storage partition in the user's device, isolated from the untrusted execution environment, to store backups, thereby protecting unauthorized access to backup data. Additionally, data to be stored in a remote data store could be encrypted using a key stored only in the trusted local storage, thus protecting even externally stored backups.

File backup is a very old topic [153, 154, 28], but most of the emphasis of such earlier work was on backup speed and space efficiency. In contrast, in our work that targets recent mobile devices that have particular characteristics, the focus is on protecting the backup mechanism itself using a modern hardware security extension (specifically TrustZone).

6.7 Summary

This chapter presented the design, implementation, and experimental evaluation of RANSOMSAFEDROID, a backup system that is protected from malware by leveraging ARM TrustZone and allows backing up files to a persistent local secure storage. This service also allows full recovery of the device's software, including the OS kernel, in case it was compromised.

Chapter 7

A Mobile Device Trust Framework

This chapter presents a software framework called MobTrust, that facilitates the development of security services to increase trust on mobile devices, i.e., to increase the accepted dependence that users can have on the security properties of such devices. *Software frameworks* are designed to simplify development of software applications for a specialized application domain by providing a reusable design and implementation [155, 156]. These frameworks include a variety of components, such as APIs, compilers, libraries, and tools to enable application development in an effective way.

The MobTrust framework aims to simplify the implementation of security services by providing components for detecting the trustworthiness of mobile devices, i.e., the level of trust we can have on mobile devices, e.g., to ensure that access to security sensitive data can only be performed in a trusted state. The design and implementation of MobTrust is based on our experience in designing and implementing the security services we present in the previous chapters (Chapters 3-5).

MobTrust provides a collection of reusable components that allow evaluation of the security status of apps, mobile OS (e.g., Android), and other system components that are vulnerable to attacks. Security services can be

implemented by instantiating the MobTrust components depending on the functionality to be supported by the services, but not necessary to include all the framework components, only a subset of them. MobTrust leverages ARM TrustZone to protect the services from apps, mobile OS (e.g., Android), and malware.

The MobTrust framework can be used by external service providers such as financial institutions, such as banks, that run mobile apps to provide a variety of services which perform security sensitive operations, such as on-line banking and mobile payments. Such service providers can develop a security service using the MobTrust framework to check the trustworthiness of a mobile device (i.e., to determine if the device has been compromised) *before* they provide a service to the device. This better protects security sensitive data (e.g., identity, financial details, etc.) from being compromised by malware that infects the device.

7.1 MobTrust components

MobTrust includes several components, both in the *normal* and *secure worlds* (as shown in Figure 7.1).

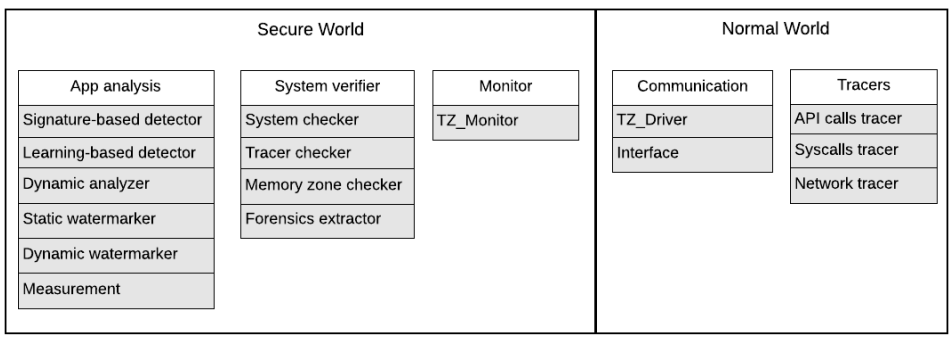

Figure 7.1: MobTrust components.

The components are categorized based on their functionality into five different modules, namely *app analysis*, *system verifier*, *monitor*, *communication*, and *tracers*. Most of the modules reside in the secure world. The normal world includes *communication* and *tracers* modules. We describe the components and their corresponding APIs in the next subsections.

Figure 7.2 presents a typical security service architecture and basic interactions. The design and implementation of security services using MobTrust involves three execution environments: (1) the normal world, for running part of the services' code (*service code*) that are shipped along with the apps that are downloaded and installed from an app market (e.g., Google Play); (2) the secure world, for running the services' components (*service trustlets*) that handle security sensitive operations by using components provided by MobTrust; and (3) the external services that run the service backend code (*service backend code*).

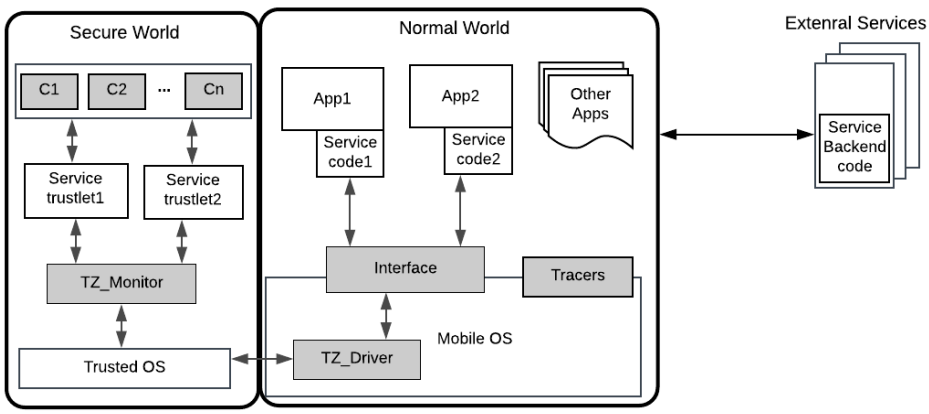

Figure 7.2: Typical service architecture and basic interactions. Grey boxes are components that are specific of the MobTrust framework.

Next, we present all the components of the framework and their APIs.

7.1.1 Tracers

The tracers module is composed of three components: *API calls tracer*, *syscalls tracer* and *Network tracer*, for monitoring Android API function calls, kernel system calls, and TCP/IP packets respectively. The API calls tracer and syscalls tracer are essentially the *API calls tracer* and *syscalls tracer* modules presented in Chapter 4.

7.1.1.1 API calls tracer

The API calls tracer intercepts and records Android API function calls made by an app using a dynamic instrumentation technique. This tracer is configured with the number of API calls to collect and/or the time to collect them. It provides the following API:

```
collect_API_calls(AppID,          // in
                  count,          // in
                  period,         // in
                  traces,         // out
                  return_code)    // out
```

This function records selected Android API functions made by an app with process ID `AppID` in a log data structure (`traces`). There is one trace per API function. The function is called with the number of traces (`count`) and/or the time `period` in seconds to collect the traces. If count is 0, then only the time period argument is valid. If the time period is 0, then only the count argument is valid. When both arguments are valid, then the process terminates when the first of either condition is satisfied. If this function call is successful, it returns `return_code` with the value of zero (SUCCESS). On an error, it returns -1 (FAILURE).

7.1.1.2 Syscalls tracer

The syscalls tracer intercepts and logs syscalls made by the target app (e.g., using the `ptrace` syscall). This component records kernel system calls made by an app with process ID `AppID` in a log data structure (`traces`). Similarly to the API calls tracer, the syscalls tracer is also configured with the number of syscalls to collect and/or the period to collect them. The same conditions are set for these arguments as for `collect_API_calls()`. It provides the following API:

```
collect_syscalls(AppID,          // in
                 count,          // in
                 period,         // in
                 traces,         // out
                 return_code)    // out
```

7.1.1.3 Network tracer

This component monitors and logs TCP/IP packets that are being received or transmitted by a network interface of a mobile device over a network to which the device is attached. This component captures and logs TCP/IP packets transmitted and received by a network interface (`interface`) of a mobile device in a log data structure (`traces`). The component is called with the number of traces (`count`) and/or the time `period` in seconds, and also other `options` to collect the packets. Similarly to API call and syscalls tracers, it can be configured with the number of packets to collect and/or the time to collect them. It also allows other options to collect packets (e.g., capture only packets of a given protocol, capture packets for specific port, capture packets from certain source and/or destination IP address, etc.). The network tracer API is:

```
collect_packets(interface,       // in
                count,           // in
                period,          // in
                options,         // in
                traces,          // out
                return_code)     // out
```

7.1.2 Communication

This module provides components that enable communication between the two worlds.

7.1.2.1 TZ_Driver

The TZ_Driver component is a kernel driver in the mobile OS, which supports cross-world calls from the normal world by leveraging the *SMC* instruction. It allocates a shared buffer that is used by the software in the normal world to pass selected data to the secure world and for the secure world to return results of calls back to the normal world.

7.1.2.2 Interface

This component provides an API that allows software running in the normal world to interact with the components in the secure world using the TZ_Driver. It has two API functions:

```
store_measurement(hash,          // in
                  return_code)   // out
```

This first function is called during a trusted boot process by the system components in the normal world including the kernel to pass a `hash` to the secure world using a shared buffer. If this call is successful, it returns `return_code` with the value of zero. On an error, it returns -1.

```
service_call(servID,          // in
             nonce,           // in
             inData,          // in
             outData,         // out
             return_code)     // out
```

This second function calls a service trustlet with service ID `servID` in the secure world. The arguments (`inData`) to be passed to the trustlet are encoded into a shared buffer. The result (`outData`, e.g., a certificate) from this call will be placed back into the buffer. If this call is successful, it returns `return_code` with the value of zero. On an error, it returns -1.

7.1.3 Monitor

This section is about the *monitor* module in the secure world that contains only one component (Figure 7.1). The *TZ_Monitor* component calculates and stores a hash of the normal world's OS in a persistent secure partition in the secure world before the secure world kernel passes control to the normal world and boots the normal world's OS. It also receives hashes of other system components running in the normal world during the course of the trusted boot process. The TZ_Monitor expects the software at each boot stage to cryptographically hash the software at the next stage, and send it the hash via `store_measurement()`.

The TZ_Monitor is also responsible for receiving a service call from an app in the normal world and determining the corresponding service trustlet in the secure world. It collects, signs results from the service trustlet, and returns them to the app in the normal world. In order to prevent attacks such as buffer overflows and other input attacks, it validates all the incoming data from the normal world.

7.1.4 System verifier

The system verifier module provides four components to do assessments of the code and data in the normal world, e.g., to check whether system components have been modified: *system checker*, *tracer checker*, *memory zone checker*, and *forensics extractor*.

7.1.4.1 System checker

This component is the same as the system verifier module in Chapter 4. It calculates a hash of the normal world's components (e.g., kernel and init) using a collision resistant hash function and compares these hashes with their corresponding hashes as measured and stored during the trusted boot process. If they do not match, then the system must have been compromised due to the collision resistance property of the hash function. This component provides the following API function:

```
verify_system(handle,         // in
              return_code)    // out
```

This function computes a hash of a system component file (`handle`, a file descriptor) running in the normal world using a hash function and compares it with this component's hash, which was stored in the secure world during a trusted boot process. If the two hashes match, it returns `return_code` with the value of 1 (MATCH); otherwise it returns 2 (NOT_MATCH).

7.1.4.2 Tracer checker

The tracer checker component is the same as the *tracer checker* module of T2Droid in Chapter 4. The tracer checker components verify the integrity of the code of the tracer components (*API calls*, *syscalls*, and *network* tracers)

running in the normal world using a hash function to validate that the traces collected by the tracer components are correct. A hash of each of the tracer components when they are in a clean state is stored in persistent storage in the secure world. This component calculates a hash of the selected module (indicated by `handle`) and compares it with the hashes that were stored. If they match, then the check is successful, otherwise it fails. This component provides the following API function:

```
verify_tracer(handle,         // in
              return_code)    // out
```

This function reads the code of a tracer component (*API call tracer*, *syscalls tracer*, or *network tracer*) running in the normal world as a file object (`handle`), calculates its hash using a hash function, and compares it with the hash of the tracer measured when the tracer was in a clean state. If they match, then it returns `return_code` with the value of 1, otherwise it returns 2.

7.1.4.3 Memory zone checker

This component checks the integrity of a memory zone in the normal world RAM (e.g., the kernel code memory pages) for modification by comparing byte-by-byte or using a hash against a memory dump collected when the system was in a pristine state.

In order to verify the integrity of the target memory zone, the starting address and length of the memory zone are required. The memory zone checker also translates the virtual address of the memory zone which resides in the normal world address space to the secure world's address space before comparing or evaluating the hash value. This component provides the following API function:

```
verify_memory_zone(start_addr,   // in
                   length,       // in
                   mode,         // in
                   return_code)  // out
```

This function reads the starting address (`start_addr`) and the size (`length`) in bytes of a memory zone, and compares it against its memory dump stored in the secure world. It uses two `modes` of operation, byte-by-byte (mode 0) or using a hash (mode 1), to verify the integrity of the target memory zone. If they match, it returns `return_code` with the value of 1. Otherwise it returns 2.

7.1.4.4 Forensics extractor

The objective of this component is to support forensics analysis of mobile devices based on data that is reliably obtained from the secure world. The component allows capturing/acquiring a full snapshot or a certain range of the normal world memory (RAM), i.e., dumping the memory of the normal world to a persistent storage area in the secure world or an external storage device. It also supports file system extraction, the acquisition of the files stored on a mobile device, including images, videos, database files (where apps store their data), system files, and logs. File system extraction enables access to data, such as passwords, phone book information, call logs, messages, emails, etc. Once the digital images/objects have been acquired, it is possible to conduct forensics analysis on these images using forensic analysis tools. This helps in digital investigation, incident response, and malware analysis. This component provides the following API functions:

```
extract_memory(start_addr,   // in
               length,       // in
               path,         // in
               format,       // in
               return_code)  // out
```

This function reads the start address (`start_addr`) and the size (`length`) of memory region (either a full or a certain memory zone in the normal world RAM), and writes it on an external storage using a filename (`path`) in a given file `format` (e.g., .txt and .dd) recognized by forensics analysis tools.

```
extract_files(SRC_DIR,       // in
              DES_DIR,       // in
              return_code)   // out
```

This function (recursively) copies files in the normal world from a given source director (`SRC_DIR`) into a destination directory (`DES_DIR`) in an external storage.

7.1.5 App analysis

The app analysis module is composed of five components for detecting whether an app is malicious or not: *learning-based detector*, *signature-based detector*, *dynamic analyzer*, *static watermarker*, *dynamic watermarker*, and *measurement*. The learning-based and signature-based detectors are the *static analysis* mechanisms implemented in the app analysis module that was presented in Chapter 3. The static watermarker, dynamic watermarker, and measurement components are the same as the *watermarking* and *measurement* modules presented in Chapter 5.

7.1.5.1 Signature-based detector

The signature-based detector looks for distinctive patterns or signatures (e.g., malicious instruction sequence used by malware) in the code of an app. It provides the following API functions:

```
init(handle,       // in
    bytecode,      // out
    return_code)   // out
```

The `init()` function unpacks the APK file (specified by the file `handle`) of an app and returns the `bytecode` (*.dex*) of the app.

```
generate_app_signature(bytecode,      // in
                       sign_app,      // out
                       return_code)   // out
```

The `generate_app_signature()` function takes the `bytecode` file of an app and converts it into a signature (`sign_app`) (e.g., a binary sequence/pattern or a hash etc.).

```
detect(sign_app,       // in
       sign_db,        // in
       return_code)    // out
```

The `detect()` function compares the signature of an app (`sign_app`) created by `generate_app_signature()` against the signatures in a database (`sign_db`) of known malware families to detect malware with similar code. If a match is found, it returns `return_code` with the value of 1, otherwise it returns 2.

7.1.5.2 Learning-based detector

This component implements a learning-based detection mechanism that utilizes machine learning algorithms on benign and malware samples to generate the

learning patterns, which can subsequently be used for detecting if an app is malware. This process was explained in Chapter 4. The component provides the following API functions:

```
init(handle,      // in
    metadata,     // out
    bytecode,     // out
    return_code)  // out
```

The `init()` function unpacks the APK file (`handle`) of an app, and extracts its `metadata` and `bytecode`.

```
create_vector(features,      // in
              metadata,      // in
              bytecode,      // in
              vector,        // out
              return_code)   // out
```

The `create_vector()` function constructs a feature vector (`vector`, i.e., a numerical representation of features) of selected features (`features`) of an app statically extracted from its `metadata` and `bytecode`.

```
detect(classifier,    // in
       vector,        // in
       return_code)   // out
```

The `detect()` function checks an app with `vector` for being malware using a trained machine learning `classifier` (e.g., with an integer index, such as 0 for kNN and 1 for Naive Bayes). If the app is malicious, it returns `return_code` with the value of 1, otherwise it returns 2.

7.1.5.3 Dynamic analyzer

This components is the same as the detector module in T2Droid (discussed in Chapter 4). It receives traces of Android API function calls and kernel

syscalls made by the app being checked, transforms them into a vector of features, and detects whether the app is malicious or benign using a machine learning classifier. There is one feature per API function and per syscall. It has the following API functions:

```
create_vector(traces,         // in
              mode,           // in
              vector,         // out
              return_code)    // out
```

The `create_vector()` function builds a feature vector (`vector`) by taking traces of API function calls or system calls as input. The function uses two `modes` of representation. The feature values might be binary (mode 0) (i.e., 1 if the app made that call, otherwise 0) or an integer equal or greater than zero (mode 1), corresponding to the number of times the call was issued by the app (i.e., the frequencies of occurrence of the call).

```
detect(classifier,    // in
       vector,        // in
       return_code)   // out
```

The `detect()` function predicts whether an app is malicious or benign given the vector by using a trained machine learning `classifier`. If the app is malicious, it returns `return_code` with the value of 1, otherwise it returns 2.

7.1.5.4 Static watermarker

The *static watermarker* implements a static watermarking scheme that verifies the authenticity and the integrity of an app using a watermark represented by the values of particular bytes in the app bytecode. Further details of the watermark and verification process were given in Chapter 5. This component has the following API functions:

```
init(handle,      // in
    bytecode,     // out
    return_code)  // out
```

The `init()` function unpacks the APK file (`handle`) of an app and returns the `bytecode` (.dex) file of the app.

```
verify(bytecode,         // in
       watermark_data,   // in
       return_code)      // out
```

The `verify()` function performs an app's authenticity by reading the watermark from the target app `bytecode` and comparing it with the static watermark data (`watermark_data`, a file handle) of the original app stored in a persistent partition reserved in the secure world. If they match, it returns `return_code` with the value of 1, otherwise it returns 2.

7.1.5.5 Dynamic watermarker

Similar to the *static watermarker* component, the *dynamic watermarker* component is also used to verify if an app is authentic or genuine by considering the syscall traces of an app (i.e., sequence of syscalls made by the app) as a watermark. It performs verification by comparing syscalls traces made by the target app with the syscalls traces of the genuine app stored in a persistent partition reserved in the secure world. This verification process was explained in Chapter 5.

```
convert_traces(traces,       // in
               sequence,     // out
               return_code)  // out
```

The `convert_traces()` function converts syscall traces (`traces`) of an app to a sequence of letters (`sequence`).

```
verify(sequence,         // in
       watermark_data,   // in
       result,           // out
       return_code)      // out
```

The `verify()` function performs authenticity verification by comparing the traces of the target app (`sequence`) running in the normal world with the reference traces/dynamic watermark data (`watermark_data`, a file handle) of the original app. If they match, it returns `result` with the value of 1, otherwise it returns 2.

7.1.5.6 Measurement

This component verifies the integrity of an app based on its hash value. It calculates the hash of an app using a collision-resistant hash function and compares it against the hash value of the original app stored in a persistent partition reserved in the secure world. If the two hash values do not match, the app has been modified.

```
init(handle,      // in
    bytecode,     // out
    return_code)  // out
```

The `init()` function unpacks the APK file (`handle`) of an app and returns the `bytecode` (.dex) file of the app.

```
verify(handle,        // in
       measurement,   // in
       result,        // out
       return_code)   // out
```

The `verify()` functions computes a hash of the target app bytecode (`handle`) and compares it against the hash (`measurement`, a file handle) of original app

stored in a persistent partition in the secure world. If the two hashes match, it returns `result` with the value of 1. Otherwise it returns 2.

7.2 Building services

The purpose of MobTrust is to simplify development of security services, allowing developers to dedicate their efforts to the requirements of the services instead of spending time implementing them from scratch. This increases the reliability of the services and reduces the programing & testing effort. In order to build a security service using MobTrust, the service has to be partitioned into three components:

service code, *service code*, running as part of an app in the normal world;

service trusted code, *service trustlet*, implementing most of the service's functionality by using the MobTrust components running in the secure world; and

backend service code, *service backend code* running as part of the service that runs in a remote server.

As depicted in Figure 7.2, the *service code* in the normal world interacts with the *service trustlet* in the secure world by using the *interface*'s API. The main components of the services are *service trustlets*. They are responsible for instantiating MobTrust components in the secure world to perform security evaluation of a mobile device (e.g., verifying the integrity of the mobile OS). We assume *service trustlets* come pre-installed with the device. We also assume that a service provider validates and signs the *service trustlets* before they are installed into the device.

Every *service trustlet* is assigned a unique identity number (*servID*), that identifies each service installed on the device. When the app with *service code* starts running, it contacts the entity to which it has to provide information about the security status of the device and obtains a nonce (to protect from replay attacks). Then, if the service uses traces, the app starts the tracers and obtains traces using the tracer components' APIs. Next, the app send a request with *servID* to the *TZ_Monitor*. The *TZ_Monitor* uses the *servID* to call the corresponding *service trustlet* in the secure world. The app uses the *interface*'s API to pass the *servID* together with traces and the nonce to the *TZ_Driver*, which passes them to the *TZ_Monitor* using a shared buffer. This process is similar to calling a remote procedure call using the `service_call()` API function of the *interface* component.

The *TZ_Monitor* calls the *service trustlet* with service ID *servID*. The *service trustlet* performs the security evaluation of the device by using MobTrust components (*C1*, *C2*,..*Cn*) and returns the result to the app in the normal world. The app finally sends the results to the entity that runs the service backend code (*service backend code*).

The development of security services using the MobTrust framework involves three basic steps. In the first step, developers have to identify the purpose of the service, i.e., what the service should do. This step is critical to define & understand the features & functionalities of the target service and to identify the parties involved in the service. The second step is to select the relevant components provided by the MobTrust framework based on the requirements of the service identified in the first step. The final step is to implement the missing components of the service, i.e., the service components that will be run in the normal and secure worlds, and also the part of the service that will be run in a backend server.

7.3 Use cases

In this section, we show the capability of the MobTrust framework by instantiating existing security services using the MobTrust framework.

7.3.1 Posture assessment

As discussed in Chapter 3, DroidPosture aims to detect and report the trustworthiness of mobile devices by analyzing the apps installed on the device and the kernel using two classes of assessment modules - app analysis and kernel analysis - and implements two examples of each: *signature-based detector*, *learning-based detector*, *syscall table checker*, and *kernel integrity checker*. DroidPosture is protected from malware by leveraging the TrustZone extension and running in the secure world. DroidPosture provides external services with a security posture report containing information about the security posture of the device to verify the device has not been compromised.

MobTrust-DroidPosture: We describe a MobTrust version of DroidPosture called MobTrust-DroidPosture. MobTrust-DroidPosture includes components of MobTrust both in the normal and secure worlds. The normal world runs *interface* and *TZ_Driver* for an app with *service code* (*MobTrust_DroidPosture_code*) running in the normal world to interact with the corresponding *service trustlet* (MobTrust_DroidPosture_trustlet) in the secure world. The secure world includes *TZ_Monitor*, *signature-based detector*, *learning-based detector*, and *system checker* components.

In order to perform a security posture assessment, MobTrust_DroidPosture_code contacts and obtains a nonce from the external service where the *service backend code* (MobTrust_DroidPosture_backend_code) is running. The app sends the *servID* together with the nonce and requests for security posture data by

invoking the *interface*'s `service_call()` function. The *TZ_Monitor* in the secure world receives this request and calls the MobTrust_DroidPosture_trustlet.

The MobTrust_DroidPosture_trustlet performs a security posture assessment (i.e., evaluates the security status of apps and the kernel) using *signature-based detector*, *learning-based detector*, *syscall table checker*, and *system checker* components, via the APIs of these components. The MobTrust_DroidPosture_trustlet returns the result back to *TZ_Monitor*. The *TZ_Monitor* creates a certificate containing the security posture data and nonce, and sends it to the MobTrust_DroidPosture_backend_code via the app in the normal world. The MobTrust_DroidPosture_backend_code verifies the nonce and the signature (using the public key certificate of the MobTrust-DroidPosture instance). This means that the MobTrust_DroidPosture_backend_code has to already have a copy of the trustlet instance's public key.

7.3.2 Dynamic analyzer

Chapter 4 presented T2Droid. T2Droid that performs dynamic (runtime) analysis of an app running on a mobile device to detect whether the app is malicious or not using traces of Android API function calls and kernel system calls performed by the app. Two tracer modules (API call tracer and syscalls tracer) are run in the normal world to record trace data of the target app. The detector module in the secure world receives the trace data and performs detection by implementing a machine learning classifier.

MobTrust-T2Droid: We describe a MobTrust version of T2Droid called MobTrust-T2Droid. The normal world runs *interface*, *TZ_Driver*, *API call tracer*, and *syscalls tracer* components. The secure world runs *TZ_Monitor* and *dynamic analyzer* components. An app with *service code* (MobTrust_T2Droid_code) collects traces of API function calls and kernel system calls using the *API call*

tracer and the *syscalls tracer* APIs, respectively. The MobTrust_T2Droid_code sends traces together with *servID* to the *TZ_Monitor* in the secure world, which calls the corresponding *service_trustlet* (MobTrust_T2Droid_trustlet) and passes it the traces.

In order to perform detection, MobTrust_T2Droid_trustlet transforms traces into a vector of features using `create_vector()` function of the *dynamic analyzer* component. It classifies this vector as malicious or benign app by using the *dynamic analyzer*'s `detect()` function.

7.3.3 Authenticity detection

TruApp was presented in Chapter 5. TruApp verifies if an app running on a mobile device is authentic. To do so, it implements static watermarking, dynamic watermarking, and measurement (i.e., cryptographic hashes over code) techniques in the secure world isolated from the mobile OS, apps and malware by leveraging the ARM TrustZone security extension. For dynamic watermarking, it runs syscall tracer in the normal world to obtain traces of kernel system calls made by the app being checked. To perform authenticity verification of an app, TruApp extracts the watermarks and/or the measurements of the app and check if they match with the information in the *verification key* (VK). TruApp provides a signed certificate to an external device or service to verify that the app is authentic. Further details on the verification process were explained in Chapter 5.

MobTrust-TruApp: We describe a MobTrust version of TruApp called MobTrust-TruApp. MobTrust-TruApp includes *interface*, *TZ_Driver*, and *syscalls tracer* components in the normal world. The secure world runs *static watermarker*, *dynamic watermarker*, and *measurement* components to

implement watermarking and measurement techniques to assess the authenticity of apps.

An app with *service code* (MobTrust_TruApp_code) contacts and obtains a nonce from its backend code (MobTrust_TruApp_backend_code) running in an external service. For dynamic watermarking, MobTrust_TruApp_code calls the *syscalls tracer*'s API to collect traces of kernel system calls made by the apps and sends *TZ_Monitor* in the secure world the trace data together with the nonce by invoking the `service_call()` function of *interface*. The *TZ_Monitor* calls the corresponding *service trustlet* (MobTrust_TruApp_trustlet) and passes it the trace data.

The MobTrust_TruApp_trustlet performs authenticity verification of the app by using the *static watermarker*, *dynamic watermarker*, and *measurement* components of MobTrust. If these all were successful, it returns a signed certificate to the *MobTrust_TruApp_backend_code* via the app to verify that the app is authentic.

7.4 Summary

This chapter presented a mobile device software framework that facilitates development of security solutions for increasing trust on mobile devices by leveraging the ARM TrustZone extension. The framework simplifies the implementation of security services by providing a set of reusable components. We demonstrated the capabilities of the framework by describing existing security services.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

To summarize the work in this thesis, we addressed the problem of unprotected security mechanisms, app repackaging, mobile ransomware attacks, and reliable system recovery, and proposed a method to improve the security of mobile devices. Our main contributions were as follows. Regarding the problem of unprotected security solutions, we designed and implemented security services that enable detection of intrusions in mobile devices by using a variety of assessment mechanisms, e.g., static & dynamic analysis of apps, and detection of kernel rootkits. Regarding the problem of app repackaging, we implemented a service that ensures an app is authentic or was not modified by an unauthorized party, by verifying the authenticity and the integrity of the app using static watermarking, dynamic watermarking, and measurements. Regarding mobile ransomware attacks and reliable system recovery, we implemented a service that allows recovering files or system components after intrusions (e.g., mobile ransomware attacks, damage caused by system errors, etc). Finally, most of the contributions are combined to develop a software framework that facilitates the development of security services for mobile devices. This

framework provides components with their corresponding APIs and enables developers to easily and quickly implement security services by invoking these components.

In general, the contribution of this thesis can be summarized in two parts: intrusion detection and recovery system. Regarding the first part, we presented security solutions that are able to detect intrusions at the app and kernel level, and that run protected from the OS, apps, and malware. To this end, three services, DroidPosture, T2Droid, and TruApp, were implemented on a real hardware that emulates the mobile devices.

- **DroidPosture**: evaluates/assesses the posture of a mobile device, i.e., the security status of the mobile OS (e.g., Android) and apps running on the device, by using a set of assessment mechanisms and reports the posture information to external service providers.
- **T2Droid**: detects if Android apps contains malware by analyzing their runtime behaviors (traces of API function calls and syscalls made by an app).
- **TruApp**: detects the authenticity and the integrity of Android apps running on a mobile device by using a set of watermarking and measurement techniques.

Regarding the second part, we presented a secure backup/recovery service for mobile devices that allows users to trust they will not lose their data.

- **ransomSafeDroid**: allows to perform backups of files periodically and a partial or full system recovery. We implemented a prototype of the service, ransomSafeDroid, on a hardware board.

8.2 Future work

In this section, we present the directions of future work.

8.2.1 Checking the integrity of peripheral devices

As part of future work, we plan to extend the implementation of the MobTrust framework with additional security components that enable verification of the integrity of peripheral devices (e.g., network card, graphics card, mouse, keyboard, and other I/O devices) that can be connected to mobile devices.

The boot sequences of most SoC designs start by executing a ROM-based bootloader (e.g., BIOS) which is responsible for initializing all peripheral devices and populating their configuration parameters (e.g., Input Output Memory Management Unit (IOMMU) configuration parameters, such as the base address). Device drivers read these configurations to determine what resources (e.g., memory-mapped locations) have been assigned to these devices. However, attackers with ring 0 privilege or that can exploit the vulnerabilities of the I/O device's firmware, can modify the I/O device configurations, e.g., they can relocate the device's memory by changing its base address. Furthermore, attackers may use peripherals that are created on purpose to launch a variety of attacks, such as USB-based attacks [157]. To address these problems, we plan to implement security components in the secure world, that can be part of the MobTrust framework and that allow verification of the integrity of peripheral devices to ensure the security of the apps that rely on these devices. One possible technique would be to check the cryptographic hash of the device's configuration parameters or its firmware. The verification process may take place during data input/output operations to these devices or at any time while they are connected to the target machine.

8.2.2 Secure function deployment in Network Function Virtualization

Internet service providers offer additional services and network functions such as network address translation (NAT), firewall, domain name service (DNS), and others, to their customers based on the Infrastructure as a Service (IaaS) model. For this purpose, service providers are beginning to use Network Function Virtualization (NFV) to design, deploy and manage network functions traditionally run on proprietary, dedicated hardware. NFV allows network functions to run on virtualized servers as virtual machines using standard virtualization technology instead of installing expensive proprietary hardware for each network function. This approach increases network flexibility and scalability, and reduces hardware costs. Service providers can purchase inexpensive switches, storage, and servers to run virtual machines that perform network functions. If a customer wants to add a new network function, the service provider can simply create a new virtual machine to perform that function. In the future, we will look at leveraging ARM TrustZone to control and secure deployment and upgrade of network functions in an NFV environment. We will also consider the use of TrustZone to verify the integrity of the network functions running in NFV servers in order to determine if they are modified in an unauthorized way.

8.2.3 Secure container

Finally, we would also like to use ARM TrustZone to design and implement container-based virtualization, such as Linux containers (e.g., LXC) to protect the confidentiality and integrity of app data from accesses by unauthorized parties, not only from other containers but also from higher privileged system software such as the OS kernel, by running the container in the secure world.

Bibliography

[1] IDC. Smartphone OS market share, 2017 Q4. `http://www.idc.com/promo/smartphone-market-share/os`, 2017. 1

[2] Michael C Grace, Wu Zhou, Xuxian Jiang, and Ahmad-Reza Sadeghi. Unsafe exposure analysis of mobile in-app advertisements. In *Proceedings of the 5th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, pages 101–112, 2012. 1, 20, 21

[3] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative Android markets. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium*, pages 5–8, 2012. 1, 20, 21, 70

[4] Yajin Zhou and Xuxian Jiang. Dissecting Android malware: Characterization and evolution. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, pages 95–109, 2012. 2, 3, 20, 22, 30

[5] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing inter-application communication in Android. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, pages 239–252, 2011. 2

[6] Zhi Xu, Kun Bai, and Sencun Zhu. Taplogger: Inferring user inputs on smartphone touchscreens using on-board motion sensors. In *Proceedings of the Fifth ACM Conference on Security and Privacy in Wireless and Mobile Networks*, pages 113–124, 2012. 2

[7] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. Privilege escalation attacks on Android. In *Proceedings of the 13th International Conference on Information Security*, pages 346–360. 2011. 2, 21

[8] Claudio Marforio, Hubert Ritzdorf, Aurélien Francillon, and Srdjan Capkun. Analysis of the communication between colluding applications on modern smartphones. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 51–60, 2012. 2, 21, 44

[9] William Enck, Machigar Ongtang, and Patrick McDaniel. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, pages 235–245, 2009. 2, 22, 23, 25, 26

[10] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, pages 627–638, 2011. 2, 22, 23

[11] Vitor Monte Afonso, Matheus Favero de Amorim, André Ricardo Abed Grégio, Glauco Barroso Junquera, and Paulo Lício de Geus. Identifying Android malware using dynamically obtained features. *Journal of Computer Virology and Hacking Techniques*, 11(1):9–17, 2015. 2, 23, 88

[12] Dan Su, Wei Wang, Xing Wang, and Jiqiang Liu. Anomadroid: Profiling android applications' behaviors for identifying unknown malapps. In *Trustcom/BigDataSE/ISPA, 2016 IEEE*, pages 691–698, 2016. 2, 23, 88

[13] Zimin Lin, Rui Wang, Xiaoqi Jia, Shengzhi Zhang, and Chuankun Wu. Classifying Android malware with dynamic behavior dependency graphs. In *Trustcom/BigDataSE/ISPA, 2016 IEEE*, pages 378–385, 2016. 2, 23, 88

[14] Man-Ki Yoon, Negin Salajegheh, Yin Chen, and Mihai Christodorescu. Pift: Predictive information-flow tracking. In *Proceedings of the 21st ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 713–725, 2016. 2, 23, 88

[15] Neal Leavitt. Mobile phones: the next frontier for hackers? *IEEE Computer*, 38(4):20–23, 2005. 2, 88, 131

[16] Adam Greenberg. Sneaky Android RAT disables required anti-virus apps to steal banking info. SC Magazine, https://www.scmagazine.com/sneaky-android-rat-disables-required-anti-virus-apps-to-steal-banking-info/article/538770/, July 2014. 2, 88, 131

[17] Kevin Kwang. Android flaw can disable, corrupt AV tools. ZDNet, http://www.zdnet.com/article/android-flaw-can-disable-corrupt-av-tools/, September 2011. 2, 88, 131

[18] Lok Kwong Yan and Heng Yin. Droidscope: seamlessly reconstructing the OS and Dalvik semantic views for dynamic Android malware analysis. In *Proceedings of the 21st USENIX Security Symposium*, pages 569–584, 2012. 2, 66

[19] Paul. Exploit code released for Android security hole. https://securityledger.com/2013/07/exploit-code-released-for-android-security-hole/, 2013. 2

[20] Hiroshi Lockheimer. Android and security. http://googlemobile.blogspot.com/2012/02/android-and-security.html, 2012. 2

[21] Xin Su, M Chuah, and Gang Tan. Smartphone dual defense protection framework: Detecting malicious applications in Android markets. In *Mobile Ad-hoc and Sensor Networks, 2012 Eighth International Conference on*, pages 153–160, 2012. 3, 20, 88

[22] Bill Horne, Lesley Matheson, Casey Sheehan, and Robert E Tarjan. Dynamic self-checking techniques for improved tamper resistance. In *ACM Workshop on Digital Rights Management*, pages 141–159, 2001. 3

[23] Amin Kharraz, William Robertson, Davide Balzarotti, Leyla Bilge, and Engin Kirda. Cutting the gordian knot: A look under the hood of ransomware attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 3–24, 2015. 3

[24] Nicoló Andronio, Stefano Zanero, and Federico Maggi. Heldroid: Dissecting and detecting mobile ransomware. In *International Workshop on Recent Advances in Intrusion Detection*, pages 382–404, 2015. 3

[25] Kaspersky. Mobile malware evolution 2016, 2016. 3

[26] D. Goodin. Ransomware app hosted in Google Play infects unsuspecting Android user. https://arstechnica.com/information-

technology/2017/01/ransomware-app-hosted-in-google-play-infects-unsuspecting-android-user/, January 2017. 3

[27] Dinesh Venkatesa. Android Marshmallow will not go soft on mobile ransomware. https://www.symantec.com/connect/blogs/android-marshmallow-will-not-go-soft-mobile-ransomware, September 2015. 4

[28] Ann Chervenak, Vivekenand Vellanki, and Zachary Kurmas. Protecting file systems: A survey of backup techniques. In *Joint NASA and IEEE Mass Storage Conference*, 1998. 4, 131

[29] Samsung. Samsung smart switch. http://www.samsung.com/nz/support/smartswitch/, 2017. 4, 130

[30] LG. LG Bridge. http://www.lg.com/us/support/product-help/CT10000026-1438110404543-preinstall-apps, 2017. 4, 130

[31] msft-mmpc. Wannacrypt ransomware worm targets out-of-date systems. https://blogs.technet.microsoft.com/mmpc/2017/05/12/ wannacrypt-ransomware-worm-targets-out-of-date-systems/, May 2017. 4

[32] Martin Pirker and Daniel Slamanig. A framework for privacy-preserving mobile payment on security enhanced ARM TrustZone platforms. In *Proceedings of the IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications*, pages 1155–1160, 2012. 5, 39

[33] Nuno Santos, Himanshu Raj, Stefan Saroiu, and Alec Wolman. Using ARM TrustZone to build a trusted language runtime for mobile applications. In *Proceedings of the 19th International Conference on Ar-*

chitectural Support for Programming Languages and Operating Systems, pages 67–80, 2014. 5, 39, 114

[34] Jan-Erik Ekberg, N Asokan, Kari Kostiainen, and Aarne Rantala. Scheduling execution of credentials in constrained secure environments. In *Proceedings of the 3rd ACM Workshop on Scalable Trusted Computing*, pages 61–70, 2008. 5, 40

[35] Kari Kostiainen, Jan-Erik Ekberg, N Asokan, and Aarne Rantala. On-board credentials with open provisioning. In *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*, pages 104–115, 2009. 5, 40, 114, 131

[36] Yossi Gilad, Amir Herzberg, and Ari Trachtenberg. Securing smartphones: a micro-TCB approach. In *arXiv preprint arXiv:1401.7444*, 2014. 5, 40, 114, 116, 131

[37] Google Inc. The Android open source project. `http://source.android.com/index.html`. 12

[38] Mark H Goadrich and Michael P Rogers. Smart smartphone development: iOS versus Android. In *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education*, pages 607–612, 2011. 12

[39] Mohd Shahdi Ahmad, Nur Emyra Musa, Rathidevi Nadarajah, Rosilah Hassan, and Nor Effendy Othman. Comparison between Android and iOS operating system in terms of security. In *Information Technology in Asia (CITA), 2013 8th International Conference on*, pages 1–4, 2013. 12

[40] Parvez Faruki, Ammar Bharmal, Vijay Laxmi, Vijay Ganmoor, Manoj Singh Gaur, Mauro Conti, and Muttukrishnan Rajarajan. An-

droid security: a survey of issues, malware penetration, and defenses. *IEEE Communications Surveys & Tutorials*, 17(2):998–1022, 2015. 20

[41] Mariantonietta La Polla, Fabio Martinelli, and Daniele Sgandurra. A survey on security for mobile devices. *IEEE Communications Surveys & Tutorials*, pages 446–471, 2013. 21

[42] Adrienne Porter Felt, Helen J Wang, Alexander Moshchuk, Steve Hanna, and Erika Chin. Permission re-delegation: Attacks and defenses. In *USENIX Security Symposium*, 2011. 21, 28, 44

[43] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. Privilege escalation attacks on Android. In *International Conference on Information Security*, pages 346–360, 2010. 21

[44] Michael Dietz, Shashi Shekhar, Yuliy Pisetsky, Anhei Shu, and Dan S Wallach. Quire: Lightweight provenance for smart phone operating systems. In *USENIX Security Symposium*, volume 31, 2011. 21, 28, 44

[45] Adrienne Porter Felt, Matthew Finifter, Erika Chin, Steve Hanna, and David Wagner. A survey of mobile malware in the wild. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, pages 3–14, 2011. 22

[46] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 259–269, 2014. 23

[47] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing inter-application communication in Android. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, pages 239–252, 2011. 23

[48] Min Zheng, Mingshen Sun, and John CS Lui. Droid analytics: a signature based analytic system to collect, extract, analyze and associate Android malware. In *Proceedins of the 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, pages 163–171, 2013. 23

[49] Androguard. `https://github.com/androguard/androguard`, 2017. 23, 61

[50] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, and Konrad Rieck. Drebin: Effective and explainable detection of Android malware in your pocket. In *Proceedings of the Symposium on Network and Distributed System Security*, 2014. 23, 65

[51] Sebastian Poeplau, Yanick Fratantonio, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. Execute this! analyzing unsafe and malicious dynamic code loading in Android applications. In *NDSS*, volume 14, pages 23–26, 2014. 23

[52] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, pages 1–6, 2010. 23, 24, 44, 88

[53] Zhemin Yang, Min Yang, Yuan Zhang, Guofei Gu, Peng Ning, and X Sean Wang. Appintent: Analyzing sensitive data transmission in Android for

privacy leakage detection. In *Proceedings of the 2013 ACM Conference on Computer and Communications Security*, pages 1043–1054, 2013. 23, 25, 88

[54] Iker Burguera, Urko Zurutuza, and Simin Nadjm-Tehrani. Crowdroid: behavior-based malware detection system for Android. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, pages 15–26, 2011. 23, 88

[55] Peter Hornyack, Seungyeop Han, Jaeyeon Jung, Stuart Schechter, and David Wetherall. These aren't the droids you're looking for: retrofitting Android to protect data from imperious applications. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, pages 639–652, 2011. 24

[56] Michael Backes, Sebastian Gerling, Christian Hammer, Matteo Maffei, and Philipp von Styp-Rekowsky. Appguard: enforcing user requirements on Android apps. In *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 543–548, 2013. 24

[57] Alastair R Beresford, Andrew Rice, Nicholas Skehin, and Ripduman Sohan. MockDroid: trading privacy for application functionality on smartphones. In *Proceedings of the 12th workshop on mobile computing systems and applications*, pages 49–54, 2011. 24

[58] Machigar Ongtang, Stephen McLaughlin, William Enck, and Patrick McDaniel. Semantically rich application-centric security in Android. In *Proceedings of the 2009 Annual Computer Security Applications Conference*, pages 340–349, 2009. 25, 26, 44

[59] Mohammad Nauman, Sohail Khan, and Xinwen Zhang. Apex: extending Android permission model and enforcement with user-defined runtime constraints. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, pages 328–332, 2010. 25, 27

[60] Mauro Conti, Vu Thien Nga Nguyen, and Bruno Crispo. CRePE: context-related policy enforcement for Android. In *Proceedings of the 13th International Conference on Information Security*, pages 331–345, 2011. 25, 27

[61] K Mueller and K Butler. Flex-P: flexible Android permissions. In *IEEE Symposium on Security and Privacy, Poster Session*, 2011. 27

[62] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, and Ahmad-Reza Sadeghi. Xmandroid: A new android evolution to mitigate privilege escalation attacks. *Technische Universität Darmstadt, Technical Report TR-2011-04*, 2011. 28

[63] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Stephan Heuser, Ahmad-Reza Sadeghi, and Bhargava Shastry. Practical and lightweight domain isolation on Android. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, pages 51–62, 2011. 29, 44

[64] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad-Reza Sadeghi, and Bhargava Shastry. Towards taming privilege-escalation attacks on Android. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium*, 2012. 29, 44

[65] Toshiharu Harada, Takashi Horie, and Kazuo Tanaka. Task oriented management obviates your onus on linux. In *Linux Conference*, 2004. 29

[66] Yury Zhauniarovich, Giovanni Russello, Mauro Conti, Bruno Crispo, and Earlence Fernandes. Moses: supporting and enforcing security profiles on smartphones. *IEEE Transactions on Dependable and Secure Computing*, 11(3):211–223, 2014. 29

[67] Stephen Smalley and Robert Craig. Security enhanced (se) android: Bringing flexible MAC to Android. In *Proceedings of the 20th Annual Network and Distributed System Security Symposium*, volume 310, pages 20–38, 2013. 30, 44

[68] Sven Bugiel, Stephan Heuser, and Ahmad-Reza Sadeghi. Flexible and fine-grained mandatory access control on Android for diverse security and privacy policies. In *Proceedings of the 22nd USENIX Security Symposium*, pages 131–146, 2013. 30

[69] Stephen Smalley, Chris Vance, and Wayne Salamon. Implementing SELinux as a Linux security module. *NAI Labs Report*, 1(43):139, 2001. 30

[70] Paul Pearce, Adrienne Porter Felt, Gabriel Nunez, and David Wagner. AdDroid: Privilege separation for applications and advertisers in Android. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, pages 71–72, 2012. 30, 31

[71] Franziska Roesner and Tadayoshi Kohno. Securing embedded user interfaces: Android and beyond. In *USENIX Security Symposium*, pages 97–112, 2013. 30

[72] Shashi Shekhar, Michael Dietz, and Dan S Wallach. AdSplit: Separating smartphone advertising from applications. In *USENIX Security Symposium*, 2012. 30, 31

[73] Yifei Wang, Srinivas Hariharan, Chenxi Zhao, Jiaming Liu, and Wenliang Du. Compac: Enforce component-level access control in Android. In *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*, pages 25–36, 2014. 31

[74] Jaebaek Seo, Daehyeok Kim, Donghyun Cho, Insik Shin, and Taesoo Kim. Flexdroid: Enforcing in-app privilege separation in android. In *Network and Distributed System Security Symposium (NDSS)*, 2016. 31

[75] Rubin Xu, Hassen Saïdi, and Ross Anderson. Aurasium: Practical policy enforcement for Android applications. In *Proceedings of the 21st USENIX Security Symposium*, pages 27–27, 2012. 31

[76] Jinseong Jeon, Kristopher K Micinski, Jeffrey A Vaughan, Ari Fogel, Nikhilesh Reddy, Jeffrey S Foster, and Todd Millstein. Dr. Android and Mr. Hide: fine-grained permissions in Android applications. In *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*, pages 3–14, 2012. 31

[77] TCG. The trusted computing group trusted platform module specification version 1.2. Published as ISO/IEC 11889 Parts 1-4 `http://standards.iso.org/ittf/PubliclyAvailableStandards/index.html`, 2015. 32, 44

[78] TCG. TPM main specification level 2 version 1.2, revision 116. https://trustedcomputinggroup.org/tpm-main-specification/, 2011. 32, 78

[79] Bryan Parno, Jonathan M. McCune, and Adrian Perrig. *Bootstrapping Trust in Modern Computers*. Springer, 2011. 33, 52

[80] MTM. TCG specification TPM 2.0 mobile reference architecture. https://trustedcomputinggroup.org/wp-content/uploads/TPM-2-0-Mobile-Reference-Architecture-v2-r142-Specification_FINAL2.pdf, 2014. 33, 44

[81] Mohammad Nauman, Sohail Khan, Xinwen Zhang, and Jean-Pierre Seifert. Beyond kernel-level integrity measurement: enabling remote attestation for the android platform. In *International Conference on Trust and Trustworthy Computing*, pages 1–15, 2010. 33

[82] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 193–206, 2003. 34

[83] ARM. ARM security technology, building a secure system using TrustZone technology. http://www.arm.com, 2009. 34, 44

[84] Tiago Alves and Don Felton. TrustZone: Integrated hardware and software security. 2005. 34, 44

[85] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative technology for CPU based attestation and sealing. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, 2013. 37

[86] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan Del Cuvillo. Using innovative instructions to create trustworthy software solutions. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, page 11, 2013. 37

[87] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. Innovative instructions and software model for isolated execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, page 10, 2013. 37

[88] Intel. Intel Trusted Execution Technology (TXT) white paper. https://www.intel.com/content/www/us/en/architecture-and-technology/trusted-execution-technology/trusted-execution-technology-security-paper.html. 37

[89] Dmitry Evtyushkin, Jesse Elwell, Meltem Ozsoy, Dmitry Ponomarev, Nael Abu Ghazaleh, and Ryan Riley. Iso-X: A flexible architecture for hardware-managed isolated execution. In *Proceedings of International Symposium on Microarchitecture (MICRO)*, pages 190–202, 2014. 38

[90] Stefan Brenner, David Goltzsche, and Rüdiger Kapitza. TrApps: Secure compartments in the evil cloud. In *Proceedings of the 1st ACM International Workshop on Security and Dependability of Multi-Domain Infrastructures*, 2017. 39

[91] Xiaolei Li, Hong Hu, Guangdong Bai, Yaoqi Jia, Zhenkai Liang, and Pratiksha Saxena. DroidVault: A trusted data vault for Android devices. In *Proceedings of the 19th International Conference on Engineering of Complex Computer Systems*, pages 29–38, 2014. 40, 88, 89

[92] He Sun, Kun Sun, Yuewu Wang, and Jiwu Jing. TrustOTP: Transforming smartphones into secure one-time password tokens. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 976–988, 2015. 40

[93] Claudio Marforio, Nikolaos Karapanos, Claudio Soriente, Kari Kostiainen, and Srdjan Čapkun. Smartphones as practical and secure location verification tokens for payments. In *Proceedings of the Network and Distributed System Security Symposium*, 2014. 40, 88, 89

[94] Sileshi D. Yalew, Gerald Q. Maguire Jr., and Miguel Correia. Light-SPD: A platform to prototype secure mobile applications. In *Proceedings of the 1st ACM Workshop on Privacy-Aware Mobile Computing*, pages 11–20, 2016. 41, 42, 88, 114, 121

[95] Xianyi Zheng, Lulu Yang, Jiangang Ma, Gang Shi, and Dan Meng. TrustPAY: Trusted mobile payment on security enhanced arm trustzone platforms. In *Computers and Communication (ISCC), 2016 IEEE Symposium on*, pages 456–462, 2016. 41

[96] Android Pay. `https://www.samsung.com/us/samsung-pay/`. 41

[97] Dongtao Liu and Landon P Cox. Veriui: Attested login for mobile devices. In *Proceedings of the 15th Workshop on Mobile Computing Systems and Applications*, 2014. 41, 47, 67

[98] He Liu, Stefan Saroiu, Alec Wolman, and Himanshu Raj. Software abstractions for trusted sensors. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, pages 365–378, 2012. 41, 47, 67, 89, 114

[99] Ahmed M Azab, Peng Ning, Jitesh Shah, Quan Chen, Rohan Bhutkar, Guruprasad Ganesh, Jia Ma, and Wenbo Shen. Hypervision across worlds: Real-time kernel protection from the ARM TrustZone secure world. In *Proceedings of the 2014 ACM Conference on Computer and Communications Security*, pages 90–102, 2014. 41

[100] Samsung. Samsung Knox security solution. https://www2.samsungknox.com/en/support/knox-workspace/white-papers, 2016. 42

[101] Apple Inc. iOS security guide. https://www.apple.com/business/docs/iOS_Security_Guide.pdf/, 2018. 42

[102] Samsung. Samsung enterprise alliance program (seap) for developers. https://seap.samsung.com/sdk, 2015. 42

[103] PCAS. Deliverable D5.3, secure data access. https://www.pcas-project.eu/index.php/deliverables/, 2015. 42

[104] PCAS. Deliverable D5.2, SPD configuration and UI. https://www.pcas-project.eu/index.php/deliverables/, 2015. 42

[105] P. Sangster, H. Khosravi, M. Mani, K. Narayan, and J. Tardo. Network endpoint assessment (NEA): Overview and requirements. RFC 5209, RFC Editor, June 2008. 47, 49

[106] Daniel V. Hoffman. *Implementing NAP and NAC Security Technologies: The Complete Guide to Network Access Control*. John Wiley & Sons, 2008. 47, 49

[107] NCSC. Trusted computer systems evaluation criteria, August 1983. 50

[108] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography*. CRC Press, Inc., 1996. 51

[109] N. S. Altman. An introduction to kernel and nearest-neighbor nonparametric regression. *The American Statistician*, 46(3):175–185, 1992. 57

[110] Genode Labs. ARM TrustZone, an exploration of ARM TrustZone technology. https://genode.org/documentation/articles/trustzone, 2014. 59, 124

[111] Androwarn. https://github.com/maaaaz/androwarn. 61

[112] Mindtrick rootkit. https://github.com/ChristianPapathanasiou/defcon-18-Android-rootkit-Mindtrick, 2014. 66

[113] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of the 10th Symposium on Network and Distributed System Security*, pages 191–206, 2003. 66

[114] Yangchun Fu and Zhiqiang Lin. Bridging the semantic gap in virtual machine introspection via online kernel data redirection. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 586–600, 2012. 66

[115] Brendan Dolan-Gavitt, Tim Leek, Michael Zhivich, Jonathon Giffin, and Wenke Lee. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 297–312, 2011. 66

[116] ENISA. Appstore security, 5 lines of defence against malware, 2011. 70

[117] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. A study of Android application security. In *USENIX Security Symposium*, 2011. 70

[118] Weka 3.8. http://www.cs.waikato.ac.nz/ml/weka/. 77

[119] Lakshmi Subramanian, Gerald Q Maguire Jr, and Philipp Stephanow. An architecture to provide cloud based security services for smartphones. In *27th Meeting of the Wireless World Research Forum*, 2011. 77

[120] M. Bishop and M. Dilger. Checking for race conditions in file access. *Computing Systems*, 9(2):131–152, 1996. 80

[121] Xposed framework. `http://repo.xposed.info/`. 81

[122] Contagio mobile. `http://contagiominidump.blogspot.pt/`. 82

[123] Monkey. `https://developer.android.com/studio/test/monkey.html`. 82, 104

[124] Android Developers. Run apps on the Android emulator. `https://developer.android.com/studio/run/emulator.html`, 2017. 83, 105

[125] Thomas Bläsing, Leonid Batyuk, Aubrey-Derrick Schmidt, Seyit Ahmet Camtepe, and Sahin Albayrak. An Android application sandbox system for suspicious software detection. In *5th International Conference on Malicious and unwanted software*, pages 55–62, 2010. 88

[126] ENISA. Algorithms, key size and parameters report – 2014. November 2014. 94

[127] ITU-T. International Telecommunication Union. ITU-T Recommendation X.509: The Directory: Public-Key and Attribute Certificate Frameworks, 2000. 94

[128] A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997. 94

[129] Ronald Cramer and Victor Shoup. Design and analysis of practical public-key encryption schemes secure against adaptive chosen ciphertext attack. *SIAM Journal on Computing*, 33(1):167–226, 2003. 97

[130] Saul B Needleman and Christian D Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453, 1970. 100

[131] Isaac Turner. seq-align: Smith-Waterman & Needleman-Wunsch alignment in C. `https://github.com/noporpoise/seq-align`, 2017. 102

[132] Connor Tumbleson and Ryszard Wisniewski. Apktool. `https://ibotpeaches.github.io/Apktool/`. 103

[133] Wenliang Du. SEEDlabs: Android repackaging attack lab. `http://www.cis.syr.edu/~{}wedu/seed/Labs_Android5.1/Android_Repackaging/`. 103

[134] Joonhyouk Jang, Hyunho Ji, Jiman Hong, Jinman Jung, Dongkyun Kim, and Soon Ki Jung. Protecting Android applications with steganography-based software watermarking. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pages 1657–1658, 2013. 110

[135] Yingjun Zhang and Kai Chen. Appmark: A picture-based watermark for Android apps. In *Proceedings of the 8th IEEE International Conference on Software Security and Reliability*, pages 58–67, 2014. 110

[136] Chuangang Ren, Kai Chen, and Peng Liu. Droidmarking: resilient software watermarking for impeding Android application repackaging. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 635–646, 2014. 110

[137] Wu Zhou, Xinwen Zhang, and Xuxian Jiang. AppInk: watermarking Android apps for repackaging deterrence. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, pages 1–12, 2013. 110

[138] He Sun, Kun Sun, Yuewu Wang, Jiwu Jing, and Haining Wang. TrustICE: hardware-assisted isolated computing environments on mobile devices. In *Proceedings of the 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 367–378, 2015. 110

[139] Pablo Anton Del Pino, Antoine Monsifrot, Charles Salmon-Legagneur, and Gwenaël Doërr. Secure video player for mobile devices integrating a watermarking-based tracing mechanism. In *Proceedings of the 11th IEEE International Conference on Security and Cryptography*, pages 1–8, 2014. 110

[140] Charles Manning. Yaffs: Yet another flash file system. https://yaffs.net/archives/yaffs-nand-specific-flash-file-system-introductory-article, 2002. 114

[141] Ted Tso. Android will be using ext4 starting with gingerbread. `https://thunk.org/tytso/blog/2010/12/12/android-will-be-using-ext4-starting-with-gingerbread/`, 2010. 114

[142] David Salomon. *Data compression: the complete reference*. Springer, 2004. 115

[143] John R Douceur, Atul Adya, William J Bolosky, P Simon, and Marvin Theimer. Reclaiming space from duplicate files in a serverless distributed file system. In *Proceedings of the 22nd IEEE International Conference on Distributed Computing Systems*, pages 617–624, 2002. 115

[144] Danny Harnik, Benny Pinkas, and Alexandra Shulman-Peleg. Side channels in cloud services: Deduplication in cloud storage. *IEEE Security & Privacy*, 8(6):40–47, 2010. 115

[145] rsync. `https://rsync.samba.org/`, 2017. 119

[146] Robert Lyda and James Hamrock. Using entropy analysis to find encrypted and packed malware. *IEEE Security & Privacy*, 5(2), 2007. 123

[147] Avery Pennarun. `https://github.com/bup/bup`. 124

[148] Freescale. i.MX53 multimedia applications processor reference manual. Document Number: iMX53RM Rev. 2.1, 06/2012, 2012. 124

[149] genbackupdata. `https://linux.die.net/man/1/genbackupdata`, 2010. 125

[150] Developer Guides. Auto backup for apps. `https://developer.android.com/guide/topics/data/autobackup.html`. 130

[151] Cloud Security Alliance. The notorious nine: Cloud computing top threats in 2013, February 2013. 131

[152] F. Rocha and M. Correia. Lucy in the sky without diamonds: Stealing confidential data in the cloud. In *Proceedings of the 1st International Workshop on Dependability of Clouds, Data Centers and Virtual Computing Environments*, 2011. 131

[153] Dennis G Severance and Guy M Lohman. Differential files: their application to the maintenance of large databases. *ACM Transactions on Database Systems*, 1(3):256–267, 1976. 131

[154] Russell J Green, Alasdair C Baird, and J Christopher Davies. Designing a fast, on-line backup system for a log-structured file system. *Digital Technical Journal*, 8:32–45, 1996. 131

[155] Wolfgang Pree. Meta patterns—a means for capturing the essentials of reusable object-oriented design. In *European Conference on Object-Oriented Programming*, pages 150–162. Springer, 1994. 133

[156] Dirk Riehle. *Framework design.* PhD thesis, ETH Zurich, 2000. 133

[157] Nir Nissim, Ran Yahalom, and Yuval Elovici. Usb-based attacks. *Computers & Security*, pages 675–688, 2017. 157