TOWARDS MODULAR AND FLEXIBLE ACCESS CONTROL ON SMART MOBILE DEVICES

Vom Fachbereich Informatik (FB 20)
an der Technischen Universität Darmstadt
zur Erlangung des akademischen Grades eines Doktor-Ingenieurs
genehmigte Dissertation von:

Dipl.-Inform. Stephan Heuser
aus Aachen, Deutschland

Referenten:
Prof. Dr.-Ing. Ahmad-Reza Sadeghi (Erstreferent)
Prof. N. Asokan (Zweitreferent)

Tag der Einreichung: 13. Juli 2016
Tag der Disputation: 29. August 2016

TECHNISCHE
UNIVERSITÄT
DARMSTADT

System Security Lab
Intel Collaborative Research Institute for Secure Computing
Fachbereich für Informatik
Technische Universität Darmstadt

Hochschulkennziffer: D17

Stephan Heuser:
Towards Modular and Flexible Access Control on Smart Mobile Devices, © July 2016

PHD REFEREES:
Prof. Dr.-Ing. Ahmad-Reza Sadeghi (1st PhD Referee)
Prof. N. Asokan (2nd PhD Referee)

FURTHER PHD COMMISSION MEMBERS:
Prof. Dr. Dr. h.c. Johannes A. Buchmann
Prof. Dr.-Ing. Mira Mezini
Prof. Dr. Max Mühlhäuser

Darmstadt, Germany July 2016

ABSTRACT

Smart mobile devices, such as smartphones and tablets, have become an integral part of our daily personal and professional lives. These devices are connected to a wide variety of Internet services and host a vast amount of applications, which access, store and process security- and privacy-sensitive data. A rich set of sensors, ranging from microphones and cameras to location and acceleration sensors, allows these applications and their back end services to reason about user behavior. Further, enterprise administrators integrate smart mobile devices into their IT infrastructures to enable comfortable work on the go.

Unsurprisingly, this abundance of available high-quality information has made smart mobile devices an interesting target for attackers, and the number of malicious and privacy-intrusive applications has steadily been rising. Detection and mitigation of such malicious behavior are in focus of mobile security research today. In particular, the Android operating system has received special attention by both academia and industry due to its popularity and open-source character. Related work has scrutinized its security architecture, analyzed attack vectors and vulnerabilities and proposed a wide variety of security extensions. While these extensions have diverse goals, many of them constitute modifications of the Android operating system and extend its default permission-based access control model. However, they are not generic and only address specific security and privacy concerns.

The goal of this dissertation is to provide generic and extensible system-centric access control architectures, which can serve as a solid foundation for the instantiation of use-case specific security extensions. In doing so, we enable security researchers, enterprise administrators and end users to design, deploy and distribute security extensions without further modification of the underlying operating system. To achieve this goal, we first analyze the mobile device ecosystem and discuss how Android's security architecture aims to address its inherent threats. We proceed to survey related work on Android security, focusing on system-centric security extensions, and derive a set of generic requirements for extensible access control architectures targeting smart mobile devices. We then present two extensible access control architectures, which address these requirements by providing policy-based and programmable interfaces for the instantiation of use-case specific security solutions. By implementing a set of practical use-cases, ranging from context-aware access control, dynamic application behavior analysis to isolation of security domains we demonstrate the advantages of system-centric access control architectures over application-layer approaches. Finally, we conclude this dissertation by discussing an alternative approach, which is based on application-layer deputies and can be deployed whenever practical limitations prohibit the deployment of system-centric solutions.

ZUSAMMENFASSUNG

Smartphones und Tablets sind heute zentrale Bestandteile unseres privaten und beruflichen Alltags. Diese Geräte sind in eine Vielzahl von Internetdiensten eingebettet und ermöglichen Anwendungen den Zugriff auf und die Verarbeitung von sicherheits- und datenschutzkritischen Informationen. Eine Vielzahl von Sensoren, angefangen bei Kameras und Mikrofonen bis hin zu Geopositions- und Beschleunigungssensoren, erlauben diesen Anwendungen und den dazugehörigen Internetdiensten tiefe Einblicke in das Nutzerverhalten. In vielen Branchen gelten Smartphones und Tablets zudem als wichtige Arbeitsmittel.

Es ist daher kaum überraschend, dass Smartphones und Tablets heute ein besonders beliebtes Ziel für Angreifer darstellen. Dementsprechend hat die Anzahl von bösartigen und den Datenschutz gefährdenden Anwendungen beständig zugenommen. Das Erkennen und Entschärfen von derartigem bösartigen Verhalten ist heute ein wesentlicher Schwerpunkt der Forschung im Bereich sicherer mobiler Systeme. Dem Betriebssystem Android wird dabei aufgrund seiner Popularität und seines Open-Source Charakters besondere Aufmerksamkeit gewidmet. Die IT-Sicherheitsforschung hat die Architektur des Betriebssystems, bekannte Angriffsvektoren und Schwachstellen genauestens untersucht und eine Reihe von sinnvollen Sicherheitserweiterungen entwickelt. Trotz der unterschiedlichen Ziele dieser Erweiterungen basieren die meisten von ihnen auf spezifischen Modifikationen des Betriebssystems und seines Zugriffskontrollmodells. Diese Ansätze sind jedoch nicht generischer Natur und decken nur bestimmte Sicherheits- und Datenschutzprobleme ab.

Das Ziel dieser Dissertation ist es, durch die Entwicklung generischer und erweiterbarer Sicherheitsarchitekturen einen soliden Grundstein für spezifische Sicherheitserweiterungen zu legen. Dieser Ansatz erlaubt die Entwicklung und Integration von Sicherheitserweiterungen, ohne das Betriebssystem selbst verändern zu müssen. Um dieses Ziel zu erreichen, analysieren wir zunächst das *digitale Ökosystem* von Smartphones und Tablets und identifizieren dessen inhärente Sicherheitsprobleme. Anschließend beschreiben wir, wie die Sicherheitsarchitektur von Android diese Probleme zu lösen versucht. Weiterhin untersuchen wir bekannte Sicherheitserweiterungen und erstellen auf Basis dieses Wissens eine Anforderungsanalyse für generische und erweiterbare systemzentrische Zugriffskontrollsysteme. Wir entwickeln zwei neuartige Sicherheitsarchitekturen, welche die Umsetzung von spezifischen Zugriffskontrollmodellen mittels Sicherheitspolicies und durch die Bereitstellung einer programmierbaren Schnittstelle ermöglichen. Anhand von vielfältigen Anwendungsfällen aus den Bereichen kontextbasierte Sicherheit, dynamische Verhaltensanalyse von Anwendungen und Isolation von Sicherheitsdomänen beschreiben wir die vielseitigen Vorzüge systemzentrischer und erweiterbarer Sicherheitsarchitekturen. Zuletzt stellen wir einen alternativen Ansatz vor, welcher die Umsetzung von Zugriffskontrollrichtlinien auf Anwendungsebene erlaubt und immer dann verwendet werden kann, wenn praktische Aspekte den Einsatz von systemzentrischen Sicherheitsarchitekturen verwehren.

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor, Professor Ahmad-Reza Sadeghi, for the opportunity of pursuing a PhD in computer science in his research groups at both Fraunhofer Institute for Secure Information Technology (SIT) and TU Darmstadt. I am deeply grateful for his guidance as well as countless fruitful discussions about mobile security. In particular, I would like to thank him for establishing outstanding collaborations with remarkable IT security researchers which led to a number of interesting projects and publications in an international environment. Further, I would like to thank Professor N. Asokan from Aalto University and University of Helsinki, not only for being the second referee for this dissertation, but also for his collaboration within the Intel Collaborative Research Institute for Secure Computing, which resulted in several publications. I would also like to acknowledge Professor Claudia Eckert and Professor Michael Waidner for offering me my first position in academia at Fraunhofer SIT.

Further, I would like to thank my closest collaborators and co-authors, namely Sven Bugiel, Christoph Busold, Markus Miettinnen, Praveen Kumar Pendyala, and Jon Rios, from the System Security Lab at TU Darmstadt, Alexandra Dmitrienko and Bhargava Shastry from Fraunhofer SIT, as well as William Enck and Adwait Nadkarni from North Carolina State University, for their collaboration on interesting research projects and their work on the publications that guide this dissertation. Similarly, thanks go to all remaining co-authors, namely Ammar Alkassar, Ingo Bente, Henry Carter, Lucas Davi, Kai-Oliver Detken, Günther Diederich, Gabi Dreo, Josef von Helden, Bastian Hellmann, Michael Jäger, Kari Kostiainen, Wiebke Kronz, Nicolai Kuntze, Marco Negro, Thien Duc Nguyen, Marcos da Silva Ramos, Bradley Reaves, André Rein, Elena Reshetova, Malte Ried, Carsten Rudolph, Julian Schütte, Christian Stüble, Patrick Traynor, Johannes Westhuis, and Jörg Vieweg.

Additional credits go to my colleagues Ferdinand Brasser, Christopher Liebchen, Stefan Nürnberger and Christian Wachsmann from TU Darmstadt, Kpatcha Bayarou, Andreas Fuchs, Rachid El Khayari, Ronald Marx, Jan-Peter Stotz, and Timo Winkelvos from Fraunhofer SIT as well as Patrick Koeberl, Steffen Schulz and Matthias Schunter from Intel for fruitful discussions about many aspects of mobile security. Similarly, I would like to thank all remaining colleagues for our productive time together.

My former flatmates René Palige and Nicola Wagner deserve special credit. René and I shared both ups and downs of our work in applied security research during our time together at Fraunhofer SIT. I will never forget our countless late-night kitchen table discussions about any topic imaginable.

I am deeply grateful to my parents and brothers for their unconditional support. Without them, this dissertation would certainly not have been possible.

Last but not least, I would like to thank Carolin Reitwießner for having my back for year after year.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

1

INTRODUCTION

Smart mobile devices, such as smartphones and tablets, have become an integral part of our daily lives. These devices serve as central information hubs, which access, collect, store and process vast amounts of privacy-sensitive data, ranging from contacts information, call data records to calendar entries. A rich set of sensors allows the user to take photos, record audio and video, track his geolocation and log his sportive activities. Thanks to the almost constant availability of high-speed wireless Internet connectivity these devices are embedded into cloud-based digital ecosystems of value-added services. While the use of cellular telephony and text messaging is declining, IP-based mobile communication has gained traction. Platform providers, such as Google and Apple, have acknowledged this trend and operate IP-based communication services for email, instant messaging, video telephony and data sharing. Further, convenient and free personal information management services, which provide seamless synchronization of sensitive data between desktop computers, smartphones, tablets and personal entertainment devices, persuade users to store and process sensitive personal information in the cloud.

Besides platform providers an increasing number of third-party developers offer specialized applications and corresponding Internet services for smart mobile devices. Enterprises have embraced these apps to increase the productivity of their workforce and to enable comfortable work on the go by extending the digital boundaries of the enterprise domain towards the mobile user. Financial services have identified smart mobile devices as ideal vehicles for the deployment of mobile payment schemes. Furthermore, smartphones and tablets have evolved into the primary user interface for the Internet of Things (IoT), where third-party applications and services cooperate with a wide variety of home entertainment electronics, external sensors, lifestyle products, wearables and e-health devices.

This convergence of functionality into dedicated software components located on one central mobile device distinguishes smartphones and tablets from *traditional* computing platforms, such as desktop PCs and notebooks: Applications deployed on smartphones and tablets store and process privacy- and security-sensitive assets, which are subject to the interests of different stakeholders - such as the user, app developers, enterprises and content providers. These applications further routinely interact with high-level operating system services as well as other applications to share data and solve complex tasks. Typical examples are sending geolocation coordinates selected within a maps application, such as Google Maps, to other persons via an instant messaging app; sharing photos taken with the camera app with friends using a social networking app; or forwarding sensitive contacts data selected in the contacts management app via an email client.

On one hand, this architecture of dedicated and reusable software components makes operating systems for smart mobile devices an ideal target for fine-grained access control enforcement. The popular Android operating system for example uses a permission-based access control model, where the user decides which privacy- and security-sensitive op-

erations applications are allowed to perform. On the other hand, Android's permission system is susceptible to a variety of attacks: Both malicious and privacy-intrusive applications abuse the fact that users are overburdened with security decisions [79, 80]. Design and implementation weaknesses lead to confused deputy vulnerabilities, where highly-privileged applications and system components accidentally leak capabilities to malicious apps [81, 152]. Finally, colluding malicious applications convey a false sense of security when inspected individually, but in the background operate in concert to escalate their privileges [52].

Addressing these concerns is an important aspect of both academic and industrial mobile security research today. Static and dynamic code analysis both promise to detect malicious behavior before corresponding applications are deployed on a user's device. However, related work has shown that these approaches are susceptible to code obfuscation techniques as well as logic bombs [201, 271, 139]. Sophisticated information flow control frameworks use dynamic taint analysis to trace sensitive information while it is processed on the device. Unfortunately, the existing architectures struggle to adequately address applications directly modifying volatile memory, for example via native code components [66]. Finally, related work has proposed both application-layer and system-centric enhancements to Android's security architecture, which augment the default permission-based access control model with use-case specific extensions. But while these extensions can effectively address many of the existing security issues, they are not generic and thus do not scale [34, 119, 14].

1.1 GOAL AND SCOPE OF THIS DISSERTATION

The main goal of this dissertation is to overcome these limitations by designing and implementing modular and flexible access control architectures for smart mobile devices, which serve as a solid and scalable foundation for the instantiation of use-case specific solutions. To achieve this goal, we analyze the mobile device ecosystem, which is driven by the interests of multiple stakeholders that need to be considered when designing access control architectures. We then scrutinize related work on security aspects of smart mobile devices by example of the popular open-source Android operating system and derive generic requirements for extensible access control architectures.

We apply the gained knowledge by designing and implementing generic and extensible access control architectures for smart mobile devices. Our FlaskDroid architecture [34] implements *policy-driven* access control, where use-case specific security solutions are instantiated by designing corresponding security policies. We then proceed to generalize this approach in the Android Security Modules (ASM) [119] framework, which provides a *programmable* interface for integrating use-case specific access control solutions into the Android operating system.

We use FlaskDroid and ASM to instantiate selected security extensions proposed by related work to demonstrate the flexibility of our extensible access control architectures. We then proceed to describe novel use-cases for system centric access control in more detail: ConXsense [162] uses FlaskDroid and ASM to provide context-aware access control and protects users against sensory malware while simultaneously improving device usability. DroidAuditor [120] adopts the ASM framework to observe and analyze the runtime behav-

ior of malicious and privacy-intrusive applications. Our TrustDroid architecture [33] and its commercial BizzTrust variant [229] both use mandatory access control to efficiently and effectively isolate enterprise and private applications and assets on smart-mobile devices. Finally, we discuss Xapp [36], which is an alternative application-layer approach for fine-grained access control that strikes a balance between usability, ease of deployment and security.

1.2 SUMMARY OF CONTRIBUTIONS

To summarize, the main contributions of this dissertation are as follows:

Extensible Policy-Driven Access Control on Android. We present a security architecture for *policy-driven* access control on Android. Our architecture, denoted FlaskDroid [34], extends SELinux [153]/SEAndroid [232] type enforcement towards Android's middleware and application layer. By instantiating selected security extensions proposed by related work we demonstrate the flexibility of our framework.

Modular and Programmable Access Control on Android. While our FlaskDroid architecture demonstrates that it is feasible to deploy flexible access control using a policy-driven approach there are use-cases where such a solution is still too restrictive. To address this limitation our Android Security Modules (ASM) framework [119] provides a *programmable* interface for designing and implementing system-centric access control solutions as Android applications.

Context-aware Access Control. Manually configuring access control policies to increase user privacy is a tedious and time-consuming process. Context-aware access control architectures aim to use environmental information derived from the onboard sensors to automatically configure corresponding security policies. Our ConXsense [162] framework prototypes context-aware access control on Android. It shows how system-centric security architectures, such as FlaskDroid and ASM, can enforce access control decisions depending on the perceived risk for device misuse and privacy exposure.

Application Behavior Analysis using System-centric Access Control. While the primary goal of system-centric access control architectures is to enforce security policies they can also serve as a useful tool for security analysts investigating malicious applications. Our DroidAuditor architecture [120] observes application behavior using the ASM framework [119] and generates a graph-based representation. We evaluate our approach by analyzing confused deputy and collusion attacks as well as malicious spyware applications using DroidAuditor.

Access Control for Secure Dual-Use of Smart Mobile Devices. Enterprises have identified smart mobile devices as useful tools to increase the productivity of their workforce by enabling comfortable work on the go. To effectively protect enterprise applications and assets from malicious and privacy-intrusive applications installed by the user it is desirable to confine them into isolated security domains. The award-winning [250, 230] BizzTrust [229] solution demonstrates how secure dual-use can be implemented on mobile devices while taking their inherent energy and computational resource constraints into ac-

count. BizzTrust evolved from our TrustDroid [33] mandatory access control architecture and is commercially available for selected Android-based smartphones and tablets today.

Access Control in Advanced IoT Scenarios. While it is generally desirable to enforce access control decisions using a system-centric approach there are situations where adopting system-centric access control architectures is not yet feasible due to practical limitations. Consider that the integration of enhanced system-centric access control architectures mandates the modification of the predeployed operating system. Related work has proposed alternative approaches, which either rely on inlined reference monitors (IRMs) that are integrated into application code [16, 289, 54, 53, 199], or use application-layer deputies for access control enforcement [136]. While IRMs can be bypassed by native code and break Android's same-origin policy for application updates, application-layer deputies can be an effective tool for access control enforcement whenever it is infeasible to modify the operating system. Our Xapp architecture [36] applies the concept of application layer deputies to Android in an Internet of Things scenario to enforce access control on resources shared between multiple mutually untrusted devices.

1.3 OUTLINE

This dissertation proceeds as follows: Chapter 2 introduces necessary background knowledge on hard- and software architectures for smart mobile devices, access control and in particular on Android's operating system architecture. Chapter 3 discusses related work on both offensive and defensive Android security research and derives requirements for extensible access control architectures. In Chapters 4 and 5 we show how our FlaskDroid [34] and ASM [119] security architectures address these requirements using a policy-driven and programmable approach towards extensible access control. In Chapter 6 we discuss a set of practical use-cases for fine-grained access control. In particular, we describe solutions for context-aware access control [162] (Section 6.1), application behavior analysis [120] (Section 6.2), security domain isolation [33] (Section 6.3) and secure sharing of resources between multiple devices [36] (Section 6.4). Finally, Chapter 7 concludes this dissertation.

1.4 PREVIOUS PUBLICATIONS

This dissertation is based on several previously published publications as listed below. The full list of publications published by the author of this dissertation can be found in Chapter 8.

Chapter 2 & 3

N. Asokan, Lucas Davi, Alexandra Dmitrienko, Stephan Heuser, Kari Kostiainen, Elena Reshetova, and Ahmad-Reza Sadeghi. *Mobile Platform Security*. Morgan & Claypool, 1st edition, 2013. ISBN 1627050973, 9781627050975. URL `http://dx.doi.org/10.2200/S00555ED1V01Y201312SPT009`.

Chapter 4

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*, USENIX'13. URL `https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/bugiel`.

Chapter 5

Stephan Heuser, Adwait Nadkarni, William Enck, and Ahmad-Reza Sadeghi. ASM: A Programmable Interface for Extending Android Security. In *Proceedings of the 23rd USENIX Security Symposium*, USENIX'14. URL `https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/heuser`.

Chapter 6

Markus Miettinen, Stephan Heuser, Wiebke Kronz, Ahmad-Reza Sadeghi, and N. Asokan. ConXsense – Context Profiling and Classification for Context-Aware Access Control. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, ASIACCS'14. URL `http://dx.doi.org/10.1145/2590296.2590337`.

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*, SPSM'11. URL `http://dx.doi.org/10.1145/2046614.2046624`.

Stephan Heuser, Marco Negro, Praveen Kumar Pendyala, and Ahmad-Reza Sadeghi. DroidAuditor: Forensic Analysis of Application-Layer Privilege Escalation Attacks on Android. In *Proceedings of the 20th International Conference on Financial Cryptography and Data Security (to appear)*, FC'16. URL `http://fc16.ifca.ai/preproceedings/15_Heuser.pdf`.

Christoph Busold, Stephan Heuser, Jon Rios, Ahmad-Reza Sadeghi, and N. Asokan. Smart and Secure Cross-Device Apps for the Internet of Advanced Things. In *Proceedings of the 19th International Conference on Financial Cryptography and Data Security*, FC'15, 2015. URL `http://dx.doi.org/10.1007/978-3-662-47854-7_17`.

2

BACKGROUND

This section covers necessary background knowledge on access control architectures for smart mobile devices. We start by introducing a generic security model which takes the peculiarities of smart mobile devices and the surrounding ecosystem into account. We proceed to provide a short primer on important access control terminology. Finally, we discuss the Android operating system architecture, focusing on security and access control aspects.

2.1 SMART MOBILE DEVICES - AN ABSTRACT MODEL

In the following, we will introduce a generic security model for smart mobile devices. We start by introducing important stakeholders present in the smart mobile device ecosystem. We then proceed to abstractly describe the hard- and software architecture of smart mobile devices.

2.1.1 *Stakeholders*

Resources stored on and accessed by smart mobile devices and their applications are subject to the interests of multiple stakeholders. These interests are not necessarily aligned, and several important design decisions regarding the hard- and software architecture of smart mobile devices reflect this aspect.

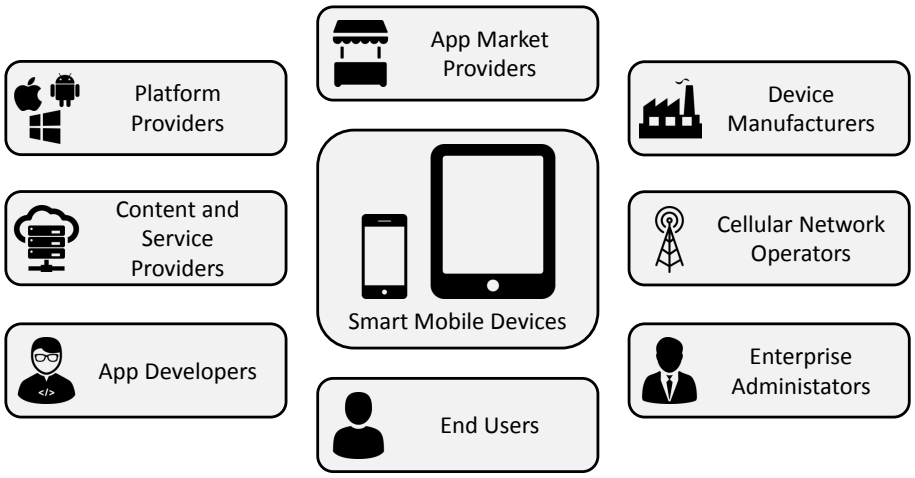


Figure 1: Smart mobile device ecosystem

Figure 1 depicts the following important stakeholders:

- **Platform Providers** develop the user-facing operating system (e.g., Android) for smart mobile devices, which is running on the application processor. Their primary goal is the protection of the integrity of the operating system software stack. Operating system integrity is a prerequisite for most security mechanisms present on smart mobile devices, as we discuss in Section 2.1.3.

- **Device Manufacturers** produce smart mobile devices and install an operating system developed by the platform provider as well as system applications on them. Device manufacturers aim to protect the authenticity, integrity and confidentiality of the device software. Adversaries who are able to break confidentiality by reverse-engineering the software stack could gain insights into the intellectual property of the device manufacturer. Compromise of software integrity and authenticity could lead to abnormal or even malicious system behavior, increasing the need for support staff and negatively affecting the manufacturer's reputation.

- **App Developers** create applications, denoted apps, for smart mobile devices. Developers aim to protect the integrity of their applications, which mandates integrity protection of the operating system, and requires user authenticity and accountability. Without application and operating system integrity users could introduce unforeseen application behavior. For example, the removal of in-app advertisement might cause a financial loss for the app developer. User authenticity and accountability are required for monetizing applications and their services. Furthermore, developers of closed-source apps aim to preserve source code confidentiality to prevent theft of their intellectual property.

- **App Market Providers** distribute apps developed by app developers to smart mobile devices. App markets typically provide software license management features, which interact with payment processors and allow app developers to sell apps to users. They further serve as software lifecycle management hubs and distribute software updates in a centralized manner. App Market Providers are primarily interested in user and device authentication as well as user accountability, which are necessary prerequisites for trading digital goods.

- **Cellular Network Operators** are responsible for the correct operation of cellular networks and services. They are interested in user and device authentication, user accountability and in the protection of the hard- and software integrity of mobile devices. Unauthorized and unaccountable users and devices within cellular networks could generate financial loss. Further, devices within the network which do not adhere to regulatory constraints could affect the availability and correct operation of the cellular network.

- **Content and Service Providers** offer specific services (e.g., VoIP telephony) and digital media (e.g., movies or music) to users of smart mobile devices. These providers require user and device authentication, user accountability as well as software and

hardware integrity. Unauthorized and unaccountable users and devices could generate a financial loss by using restricted services without reimbursement. Further, integrity-compromised devices could allow users to circumvent digital rights management (DRM) mechanisms which impose restrictions on the use of digital media assets, such as rented books or movies.

- **End Users** are primarily interested in in protecting the confidentiality and integrity of privacy- and security-sensitive assets stored on and accessed by their devices and applications, such as their personal contacts information, call logs, sensory data as well as payment information (e.g., credit card data).

- **Enterprise Administrators** integrate corporate- or privately-owned devices into an enterprise IT infrastructure. They aim to preserve the integrity and confidentiality of sensitive enterprise assets stored on and accessed by the mobile device. They further are interested in preserving software and hardware integrity of enterprise-owned devices as well as user and device authenticity and accountability.

It should be noted that the same logical entity can represent multiple stakeholders. For example, Apple Inc. represents the device manufacturer, platform and app market provider as well as the primary content provider for iOS devices. Google Inc., on the other hand, represents the platform provider, primary app market provider and primary content provider for the Android operating system.

It is obvious that the goals of these multiple stakeholders are often not aligned. For example, while content providers aim to protect their digital assets from unauthorized use, private users might be tempted to disable corresponding media playback restrictions. Enterprise administrators require software integrity. Private users might attempt to violate this integrity by gaining administrative "root" privileges, a process also denoted as "jailbreaking", in order to enhance the functionality of their devices and to lift usage constraints imposed by enterprise security policies. Addressing these diverse goals is challenging and an important motivation for modular and extensible security architectures.

2.1.2 *Hardware Architecture of Smart Mobile Devices*

Smart mobile devices are designed around energy-efficient System-on-Chips (SoCs), which condense many required hardware components into one integrated circuit. Contemporary SoCs for smartphones and tablets are highly complex and typically consist of a fabric which interconnects separate logical units. First and foremost such a SoC contains a central multicore processing unit, a graphics processor and a memory controller, which is accompanied by a limited amount of volatile and non-volatile memory. These central components are augmented by further peripherals, such as additional volatile and non-volatile memory, a microphone and speaker, display and touchscreen controllers, wireless network interfaces as well as power management circuitry via I/O interfaces.

Figure 2 abstractly depicts the hardware architecture of a modern smart mobile device. It should be noted that logical units, albeit implemented on the same SoC, are partially executed on separate microprocessor or -controller cores. For example, the baseband sub-

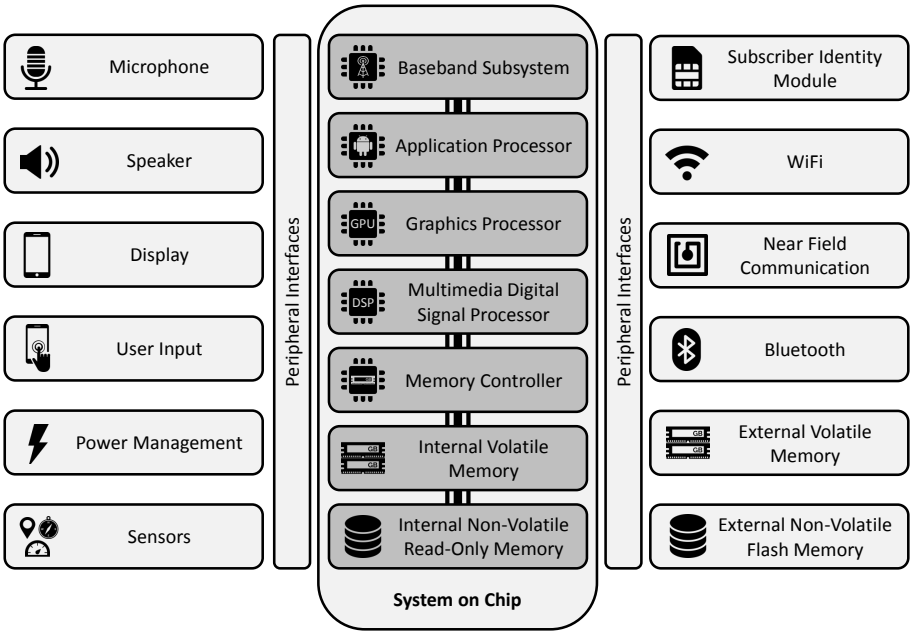


Figure 2: Hardware architecture of contemporary smart mobile devices

system, which executes the cellular networking stack, is typically implemented using a separate processor, denoted *baseband processor*. This subsystem communicates with the *application processor*, which runs the user-facing operating system (e.g., Android or iOS). This design choice is driven by the necessity to prevent the user-facing operating system from interfering with the cellular networking stack. This networking stack is subject to the interests of mobile network operators and has to comply with regulatory standards to prevent accidental or intentional negative effects on the cellular network.

Furthermore, modern SoCs for smart mobile devices usually include hardware support for the implementation of a Trusted Execution Environment (TEE). The primary purpose of a TEE is to provide an isolated code execution environment on the device, which is out of reach of the operating system running on the application processor. TEEs are for example used to perform cryptographic operations and store secret keys and data for payment and digital rights management processes in isolation. Primary beneficiaries of TEEs today are content and service providers. The underlying hardware security extensions and instruction set architectures, such as *ARM TrustZone* [7], provide the required memory isolation primitives and secure interfaces to SoC-specific I/O hardware (e.g., a TEE-aware interface to the touchscreen controller). TEE software stacks, such as *Trustonic Kinibi* [268], build upon these hardware security extensions and provide an execution environment and programming interface for isolated services.

2.1.3 *Software Architecture of Smart Mobile Devices*

The software stack of operating systems for smart mobile devices can be described as a three tier architecture, which consists of the operating system kernel layer, the middle-

ware layer and the application layer. In the following, we will describe this abstract model while focusing on important security aspects. We start by discussing the platform initialization process, which occurs when the device is powered up, and proceed with the kernel, middleware and application layer. Figure 3 depicts this abstract software architecture.

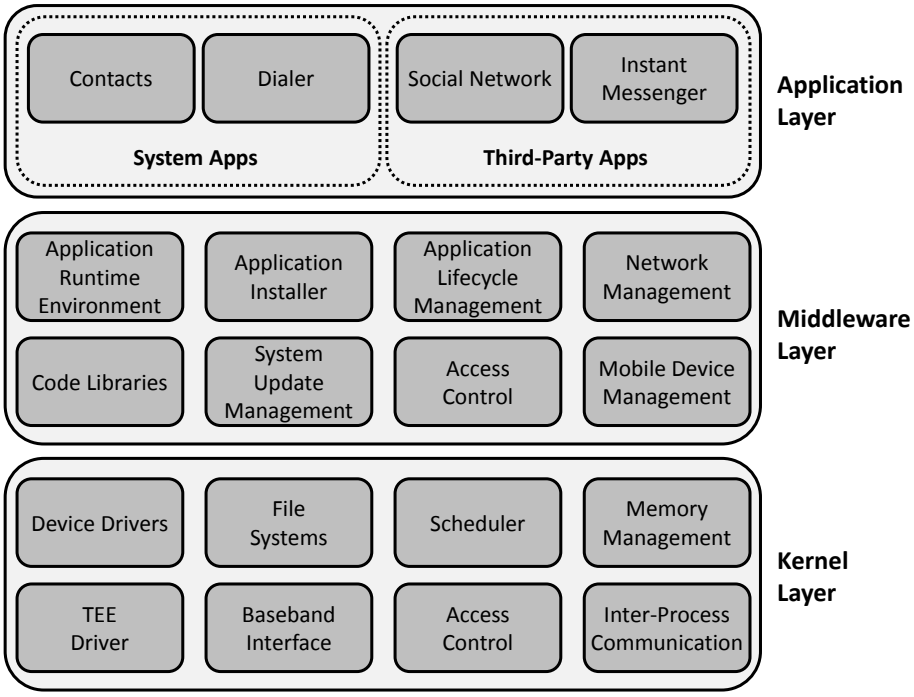


Figure 3: Software architecture of contemporary smart mobile devices

Platform Initialization. When a smart mobile device is powered up a platform initialization process, also denoted *bootstrapping* or *boot* process, initializes the hardware before handing over control to the operating system kernel. Despite ongoing efforts to standardize this procedure, such as the Unified Extensible Firmware Interface (UEFI) standard [269], bootstrapping a SoC is still a mostly vendor-specific procedure.

This bootstrapping process plays a vital role in establishing platform security. Ideally, it ensures that a device is initialized into a well-known state, in which the security mechanisms integrated into the hardware architecture and software stack operate as designed. This property is denoted *platform integrity*, and can be achieved using concepts developed by the trusted computing community - namely *secure* and *trusted boot*. Both approaches establish a *chain of trust*, in which every piece of software executed on the device first verifies the integrity of other software components before executing them.

Figure 4 depicts an abstract model for platform initialization, which approximates typical vendor-specific platform initialization procedures. In this model, the initialization process is implemented using multi-stage boot loaders, which is a standard procedure for SoCs for smart mobile devices [122]. When the SoC is powered up, it loads an initial SoC vendor-specific first-stage boot loader from a fixed address in on-chip non-volatile write-protected memory. This boot loader can typically not be modified or upgraded with-

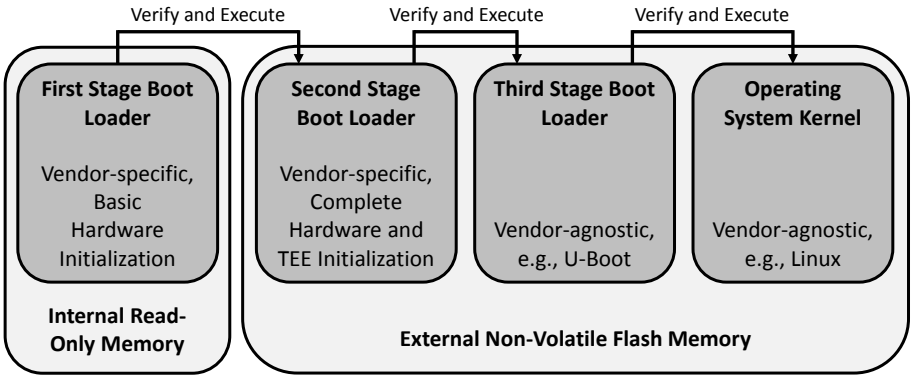


Figure 4: Platform initialization process

out exchanging the entire SoC. The first stage boot loader initializes the minimal set of hardware components required to load the second stage boot loader, for example from non-volatile rewritable flash memory. Before handing over control to the second stage boot loader it uses cryptographic mechanisms, such as cryptographic signatures or message authentication codes, to verify its integrity and authenticity. It thus ensures that the second stage boot loader, which can be updated or modified without exchanging the SoC, has not been tempered with by an adversary.

In the next step, the vendor-specific second stage boot loader completes the initialization of the SoC hardware, including the cellular radio subsystem. It further initializes the Trusted Execution Environment (TEE), ensuring its authenticity and integrity using cryptographic mechanisms. The second stage boot loader finally verifies and loads the third-stage boot loader, which typically is vendor-agnostic. Linux-based devices for example often rely on a variant of the popular U-Boot boot loader [90].

Finally, the third-stage boot loader proceeds to load the operating system kernel from non-volatile rewritable flash memory. Ideally, the third stage boot loader ensures the integrity and authenticity of the operating system kernel before handing over control. In turn, the operating system ideally verifies the integrity of each software component loaded within userland. However, many manufacturers of smart mobile devices do not extend the chain of trust beyond the initialization of the TEE, and thus allow the user to install arbitrary operating systems on his device.

Kernel Layer. The kernel of the user-facing operating system (e.g., Android or iOS) provides necessary primitives for executing multiple software components on the smart mobile device in a controlled way. In our abstract model, the operating system kernel is executed on top of the "bare-metal" hardware, which means that it has full control of the underlying SoC (with the exception of the cellular radio stack executed on the baseband processor and the TEE). It should be noted that our model does not consider virtualization, which would introduce another abstraction layer between the hardware and kernel layer. While there are approaches which employ virtualization to isolate multiple operating systems deployed by different stakeholders [18, 51, 127, 146], or to remove the need for a separate baseband processor [118], these approaches are rarely used in practice due to resource (e.g., processing power and memory) and usability constraints.

Almost all operating systems for smart mobile devices in active use today, namely Android, iOS, Windows Phone, Windows 10, Firefox OS, Tizen, Sailfish OS and Ubuntu Phone, are based on monolithic or hybrid operating system kernels (see Table 1). One exception is Blackberry OS 10, which is based on a variant of the QNX microkernel operating system [194]. It should however be noted that with the introduction of the Blackberry Priv in 2015, Blackberry is starting to lean towards Android as an alternative to the established QNX-based product line.

Name	**Primary Developers**	**Kernel**	**Design**	**Sales Market Share Q4/15 [130]**
Android	Google Inc.	Linux	Monolithic	80.7%
iOS	Apple Inc.	XNU	Hybrid	17.7%
Windows Phone	Microsoft	Windows NT	Hybrid	1.1%
Blackberry OS	Blackberry Ltd.	QNX	Microkernel	0.2%
Firefox OS	Mozilla Foundation	Linux	Monolithic	$< 0.2\%$
Tizen	Linux Foundation, Tizen Association, Samsung, Intel	Linux	Monolithic	$< 0.2\%$
Sailfish OS	Jolla Oy	Linux	Monolithic	$< 0.2\%$
Ubuntu Phone	Canonical Ltd.	Linux	Monolithic	$< 0.2\%$

Table 1: Operating system architectures and sales market share

Microkernels by design only execute a minimal set of services - most importantly memory management and task scheduling - as part of the operating system kernel in a highly privileged protection domain, and shift additional functionality into services running in less privileged protection domains [249]. In contrast, monolithic and hybrid architectures incorporate non-central operating system services, such as device and filesystem drivers as well as the networking stack, into kernelspace. Thus, from a security standpoint microkernel based architectures have a potentially smaller trusted computing base (TCB) footprint. The trusted computing base describes the hard- and software components which have to be trusted to preserve the security guarantees of a particular system. On the other hand, microkernel based operating systems are not supported by a large open-source developer community. Considering the short-lived development cycles of the underlying SoC hardware, this aspect impedes the development of microkernel-based operating systems for smart mobile devices in practice.

When comparing operating system kernels targeting smart mobile devices to kernels for desktops and servers only minor differences are apparent. Android, Firefox OS, Tizen, Sailfish OS and Ubuntu Phone use standard Linux kernels which only contain minor changes concerning energy management, inter-process communication and security. Further, the kernel layer typically provides an operating system specific hardware abstraction layer (HAL), such as the Android HAL [135], which abstracts from the SoC vendor specific hardware and device drivers.

Middleware Layer. The middleware layer is located within the userland environment of smart mobile devices and exposes a well-defined application programming interface (API) towards apps. It provides the application runtime environment, which is responsible for controlled app execution and exposes shared code libraries implementing common func-

tionality. Further, a set of standardized middleware service components are responsible for application installation and lifecycle management, software updates as well as mobile device management. These services also serve as an abstraction of kernel-layer functionalities, such as access to hardware resources and power management functionality.

This convergence of functionality into well-defined high-level services distinguishes operating systems for smart mobile devices from desktop and server operating systems and serves as a basis for security policy enforcement on the middleware layer. To this end, reference monitors located inside these service components enforce access control rules which limit operations on sensitive resources to authorized applications.

Application Layer. Applications are executed using the application runtime environment provided by the middleware layer. Generally, applications can be grouped into preinstalled *system* applications and *third-party* applications installed by the user. System applications augment the operating system API and implement additional high-level services expected to be present on smart mobile devices, such as contacts and calendar databases. Similar to operating system services implemented on the middleware layer they too enforce access control rules by means of reference monitors.

In contrast, third-party applications are installed by the user via app markets, such as Google Play Store [100] or Apple's App Store [6]. These app markets also serve as central distribution points for app updates. The platform provider can limit app distribution to authorized channels. Apple, for example, enforces that aside from specific enterprise use-cases [196] apps can only be deployed via the Apple App Store. In contrast, Google generally allows application sideloading on Android, a term describing application installation without locking users into a particular app store.

Application distribution via app stores may include an app vetting process, during which manual and automated tests are performed to check whether or not applications adhere to certain guidelines. These guidelines on one hand can be geared towards ensuring a uniform look and feel of the platform. On the other hand, these guidelines can be part of the platform security strategy. By adopting static and dynamic code analysis malicious software can to some extend be detected before applications are distributed to users. However, related work has shown that developers of malicious software actively attempt to thwart static and dynamic code analysis [201, 174]. Accordingly, these mechanisms should only augment the on-device security architecture.

2.2 ACCESS CONTROL

We now briefly discuss necessary background knowledge on access control, which is a fundamental aspect of operating system security. Access control covers aspects of authentication and authorization and abstractly describes *operations* specific authenticated entities, denoted *subjects*, are authorized to perform on other entities or resources, denoted *objects*.

2.2.1 *Access Control Matrix*

Lampson describes a formal model for access control based on *subjects* and *objects*. His model describes operations $op \in \mathsf{OP}$ subjects $s \in \mathsf{S}$ are authorized to perform on objects $o \in \mathsf{O}$ using an access control matrix [145]. Table 2 shows an example of such an access control matrix which encodes a set of access control rules.

	Object o_1	**Object** o_2
Subject s_1	read	$\emptyset$
Subject s_2	read, write	read
Subject s_3	$\emptyset$	read, write

Table 2: Example access control matrix

In common multiuser operating systems, such as Microsoft Windows, Apple OS X and Linux, subjects are generally processes operating on behalf of users. Objects are resources, services or other processes a subject performs operations on. This abstract notion of objects mandates that operations are object-specific. While generic *open*, *read* and *write* operations may apply to a variety of objects, such as files, directories and sockets, more specific operations, such as *sendMessage* or *receiveMessage* may only apply to specific communication mechanisms, such as Inter-Process Communication (IPC) channels.

A set of access control rules $AC(s, o, op) = allow|deny$ can be described using *access control lists (ACLs)* or *capabilities*. An ACL represents a column in the access control matrix and thus describes access control rules from the perspective of individual objects. For example, the ACL for Object o_1 in Table 2 is $ACL_{o_1} = \{\{s_1 : \text{read}\}, \{s_2 : \text{read,write}\}, \{s_3 : \emptyset\}\}$. In contrast, capabilities represent rows in the access control matrix and describe access control rules from the perspective of individual subjects. The capabilities of subject s_1 thus are $CAP_{s_1} = \{\{o_1 : \text{read}\}, \{o_2 : \emptyset\}\}$.

2.2.2 *Discretionary and Mandatory Access Control*

While the access control matrix defines abstract access control rules for subjects and objects, it does not define how or by whom these rules are created or which entities are authorized to modify them at runtime. Related work [219] distinguishes between two fundamentally different approaches, namely *discretionary* and *mandatory* access control, and we will describe these models in the following paragraphs.

Discretionary Access Control. In *discretionary* access control (DAC) the owner of an object, which is initially the creator of this object (or rather his process) defines access control rules for this object. Access control rules are thus defined *at the discretion* of users. Discretionary access control is commonly used as an access control mechanism for filesystems. For example, Linux as well as most other unix-like operating systems define three basic operations (read, write and execute) which apply to three groups of subjects (owner processes, processes of users of the same user group and everyone else). Further, the Linux kernel exposes interfaces to important kernel functionalities via pseudo filesystems,

which are subject to discretionary access control as well. Important examples are domain and network sockets, block and character devices as well as named pipes used for inter-process communication [154].

Mandatory Access Control. A common issue of discretionary access control is that this model cannot implement information flow control, which is a security model that aims to restrict propagation of sensitive information [211]. Allowing a subject to define access control rules for objects it creates could expose sensitive information contained in these objects to less privileged subjects. Consider for example a user who stores sensitive information not to be disclosed to others on a multiuser operating system. Discretionary access control provides no means to prevent this user from granting read privileges on this sensitive information to other users. Preventing such undesired information flows is traditionally the main objective of *mandatory* access control (MAC).

The term mandatory access control is rooted in multilevel security architectures [178], where access control rules are defined based on object sensitivity and subject clearance. In contrast, modern operating system architectures use a broader definition of mandatory access control [219]. A common denominator of these mandatory access control architectures is that a privileged entity, usually the administrator of the system, defines access control rules which allow specific subjects to perform specific operations on specific objects. These mandatory access control rules cannot be overruled by the owner of an object or any other unauthorized entity.

Mandatory access control models for modern operating systems are enforced by system-centric access control frameworks. First and foremost, they control sensitive operations within the system call interface of the kernel to userland processes. Furthermore, privileged userland applications which expose sensitive functionality to other processes enforce mandatory access control rules on IPC transactions. This is especially true for microkernel-based operating systems, which by design shift significant amounts of sensitive functionality, such as device drivers, to userland processes.

It should be noted that discretionary and mandatory access control are not mutually exclusive concepts. When both mechanisms are active access control decisions are derived by combining the results of both models: Only if both a DAC rule $AC_{DAC}(s, o, op) = allow$ and a MAC rule $AC_{MAC}(s, o, op) = allow$ exist the operation is allowed to proceed. More formally: $AC = AC_{DAC} \wedge AC_{MAC}$.

The Android operating system, which this dissertation focuses on, employs both discretionary and mandatory access control models to protect security- and privacy sensitive resources from unauthorized access. In the following section we will briefly describe this operating system based on our abstract model introduced in Section 2.1.3.

2.3 THE ANDROID OPERATING SYSTEM

Android [131] is a Linux-based operating system for smart mobile devices, such as smartphones and tablets. It has been developed by the startup Android Inc., which was acquired by Google Inc. in 2005. Android was released to the public in 2007. The core operating system is open-source software and denoted as the *Android Open Source Project (AOSP)*.

Android's underlying Linux kernel is subject to the copyleft GPLv2 license. However, the userland environment, also denoted as Android's middleware layer, consists of various open-source components released under a variety of software licenses. Notably, Android's core middleware-layer components are subject to the Apache Software license, which allows device manufacturers to distribute modified and extended Android versions without publishing the complete source code. Commercial devices distributed under the Android brand have to adhere to standards described in the Android Compatibility Definitions Document (CDD) and pass the Android Compatibility Test Suite (CTS) [95].

Device manufacturers ship commercial devices with preinstalled proprietary Google Mobile and Play Service applications. These closed-source apps connect Android devices to Google's cloud-based service infrastructure. Notable services are personal information management (Google Mail, Calendar and Contacts) as well as the social network Google Plus and the Youtube video sharing portal. Third-party applications are distributed primarily via the Google Play Store app market, previously referred to as the Android Market. Google charges a fee of USD $25.00 to become a registered Google Play developer as well as a 30% fee for every transaction processed via Google Play Store. Since 2012, Google also distributes multimedia content, such as videos, music and eBooks via Play Store. This tight binding between Google's cloud-based services and the Android operating system prompted the European Commission to launch an antitrust investigation against Google Inc. in April 2015 [72].

2.3.1 *Architecture*

The Android operating system adheres to the abstract software architecture model described in Section 2.1.3. Figure 5 depicts the Android operating system as an instantiation of our generic model. In the following, we will describe important Android-specific system components located on the kernel, middleware and application layer.

Kernel Layer. On top of the hardware platform a Linux kernel is responsible for low-level operating system services, such as memory and process management, file system and network operations. The Android Linux kernel further introduces changes primarily in the power management, logging and memory management subsystems of the mainline Linux kernel. Most importantly, the Android Linux kernel ships with the Binder Linux inter-process communication (IPC) mechanism, which is a fork of OpenBinder [184]. Binder is a service-based IPC mechanism which uses remote procedure call semantics and serves as the default inter-application communication mechanism.

The Android Linux kernel further provides important security primitives which lay the foundation of Android's security architecture. In particular, Android enforces process-level sandboxes using discretionary access control for filesystem level isolation, SELinux [153] mandatory access control to harden process sandboxes, as well as additional kernel-layer reference monitors to enforce access control on the network and Bluetooth subsystems. We will describe these mechanisms in more detail in Section 2.3.2.

It should be noted that while these changes have partially been upstreamed to the mainline Linux kernel sourcecode, their complete integration is an ongoing project. Ac-

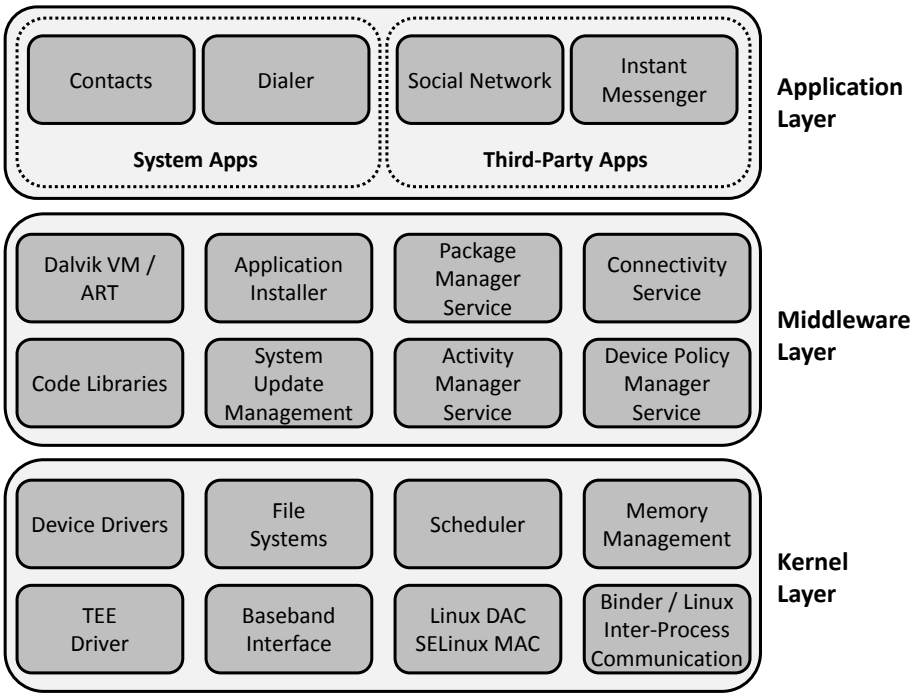


Figure 5: Android high-level architecture

cordingly, at the time of writing this dissertation the AOSP sourcecode still depends on custom versions of the Linux kernel.

Middleware Layer. In the userland environment Android's middleware layer provides a set of program libraries, such as a C standard library and a Java classpath implementation, as well as high-level operating system services. These services and libraries implement most of Android's Application Programming Interface (API), which is exposed to applications via well-defined interfaces.

For security reasons privileged security- and privacy-sensitive operations are performed beyond control of an application's process. Most of these operations are implemented in Android's highly privileged middleware-layer SystemServer process. Examples are Android's SensorService and LocationManagerService, which provide access to sensor data and location information, and CameraService, which exposes an interface to the on-device camera. From a platform security perspective, the most notable Android Services are ActivityManagerService and PackageManagerService. ActivityManagerService manages application component lifecycle and controls graphical user interfaces, while PackageManagerService is responsible for application package installation and uninstallation, as well as maintaining metadata about installed applications and their interfaces. ActivityManagerService further implements a broker and reference monitor for Binder-based IPC, which we will discuss in detail in Section 2.3.2.

Finally, the middleware layer provides a standardized execution environment for applications, either based on the Dalvik virtual machine or, in more recent Android versions, the Android Runtime Environment (ART). This execution environment as well as important

software libraries are initialized at system boot within the *zygote* process. The zygote process is then at runtime duplicated via a fork operation for every new application instance.

Application Layer. Android applications are primarily developed in Java and compiled into DEX bytecode, which is optimized for low-memory devices. On the device, this DEX bytecode is either interpreted or just-in-time compiled to native code using the Dalvik virtual machine, or ahead-of-time compiled to native code using the Android Runtime (ART), depending on the version of the Android OS [97]. This design choice makes Android applications generally platform-independent. Nonetheless, app authors can optionally implement and ship app components using platform-specific native code, for example to improve performance. Applications are packaged by the developer into application package (apk) files, which contain the DEX bytecode, optional native code components and additional resources, such as text resources or graphical user interface elements. Further, the application package contains a manifest file, which bundles application metadata, such as the application name, version number and supported Android versions.

Android applications are designed to communicate with each other and share data. To this end, third party and system applications as well as Android's middleware layer are implemented using a component model, which provides standardized interfaces for information exchange between application components. As noted before, communication between components across process boundaries is realized via the Binder inter-process communication (IPC) mechanism of the underlying Linux kernel. Application components can be grouped into four distinct categories, which we describe in the following.

- **Activities** model graphical user interfaces and control logic.
- **Services** execute tasks in the background without direct user interaction.
- **ContentProviders** implement an SQL-style interface for storing and accessing structured data. Their back end storage engines are typically implemented via SQLite databases [238].
- **Broadcast Receivers** are mailboxes for Binder IPC messages called Intents.

When an application component invokes another application component it does so by sending an Intent message to the operating system. This Intent either explicitly describes the target component and application or contains an abstract definition of the component's functionality. For example, an Intent might abstractly describe the operation of displaying a particular web page. The operating system, namely Android's ActivityManagerService and PackageManagerService, resolve the target component and invoke the component on behalf of the requesting application. In case multiple components are suited to handle the Intent, for example in case the user has installed multiple web browsers, the user is requested to choose an application via a graphical user interface.

This IPC-centric development model is a fundamental difference between standard desktop and server applications and Android applications: While the former usually have one specific program entry point, the latter have multiple entry points, namely the aforementioned application components.

2.3.2 *Security Considerations*

Overview. Android enforces a least privilege model on applications. The goal of this model is to enable fine-grained access control on security- and privacy-sensitive resources and services. To this end, Android applications as well as operating system services are executed in isolated least privilege sandboxes, and every access to privacy- and security-sensitive resources is controlled by a corresponding reference monitor. These reference monitors are implemented on all layers of the operating system, namely the kernel, middleware and application layer.

Integrity Considerations. Preserving the integrity of these reference monitors and the corresponding decision logic it is a fundamental requirement for any operating system security architecture. As such, we first need to discuss the trusted computing base (TCB) of the Android operating system. As noted before, the TCB describes the hard- and software components which have to be trusted to preserve the security guarantees of a particular architecture.

Since Android deploys reference monitors on the kernel, middleware and application layer the TCB of the Android operating system consists of the hardware platform, boot loader (chain), Linux kernel as well as large parts of Android's middleware and application layer, namely security-sensitive operating system services implemented within userland. The trusted computing community has developed a range of techniques to verify and attest the integrity of the TCB. The actual adoption of these techniques on end-user devices varies between device vendors. For example, AOSP generally provides means to verify the integrity of the TCB statically at system startup using secure and authenticated boot [103]. However, to establish a *complete* chain of trust support by the SoC vendor is required, since the platform initialization process of different SoCs varies significantly (see Section 2.1.3).

Android enforces mandatory code signing not only for operating system updates but also for system and third-party applications. However, no central public key infrastructure is used. Instead, application developers (usually) use self-signed certificates and corresponding key pairs to sign their applications before they are published. Accordingly, mandatory code signing is mainly used to enforce a same-origin policy for application updates. This same origin policy is enforced by Android's PackageManagerService and ensures that application updates can only be provided by the same app developer.

Kernel-Layer Least Privilege Application Sandboxes. Android isolates applications and operating system services using kernel-layer least privilege sandboxes, which are based on the standard Linux process model. Filesystem level isolation is implemented using the discretionary access control (DAC) model of the underlying Linux kernel. At installation time, PackageManagerService assigns an individual Linux User ID (UID) and corresponding private home directory to an application, and every process spawned by the application is executed using this UID[1]. Applications signed with the same developer key can optionally share a UID, which essentially means that these applications share a sandbox. In contrast

[1] An exception is Android's isolatedProcess mechanism, which is currently primarily used by web browsers to isolate untrusted JavaScript code from the web browser's application sandbox.

to traditional Linux distributions Android by default does not allow the user to execute processes with administrative "root" privileges.

Android's DAC-based application sandboxing mechanism is further strengthened by the adoption of the SELinux [153] kernel-layer mandatory access control architecture, also denoted as SEAndroid [232]. Since Android version 4.3 a corresponding access control policy is generated by the device manufacturer when building the Android OS for a particular device. This policy restricts system services and only allows operations they fundamentally require for correct operation to proceed. In contrast, all third-party applications deployed on standard Android distributions are subject to the same SEAndroid access control policy. Consequently, additional access control mechanisms are required to enable fine-grained user-driven control over application privileges.

Permission-based Access Control. Based on the previously introduced least privilege sandboxes Android implements an access control model which restricts access to privacy- and security sensitive operating system components and resources to authorized applications. Android's access control model uses *permissions*, which are text strings describing application capabilities. The Android operating system declares a set of fixed permissions, which protect access to operating system resources. Important examples are the `INTERNET` permission, which allows application processes to open network and domain sockets, or the `READ_CONTACTS` and `WRITE_CONTACTS` permissions, which enable application processes to access the contacts ContentProvider. Permissions are categorized into five distinct groups:

- **Normal** permissions protect privacy- or security-insensitive resources and components which reside outside of the application sandbox.
- **Dangerous** permissions protect privacy- or security-sensitive resources and components which reside outside of the application sandbox.
- **Signature** permissions protect resources and components which should only be accessible by applications signed with the same developer private key as the application declaring the permissions.
- **System** permissions protect resources and components designed for access by operating system components only.
- **SystemOrSignature** permissions combine the **System** and **Signature** protection level to protect resources and components which should only be accessible by preinstalled system applications or by applications signed with the same key as the application declaring the permissions.

Application developers declare at development time which permissions their apps require and list them in the application manifest file. Until Android version 6.0, permissions were assigned to applications statically at installation time and could not be revoked at runtime. Thus, users had to decide between either granting all requested permissions to an app, or not installing this app. This aspect changed with the release of Android 6.0, where users can en- and disable individual permissions at runtime. Accordingly, app developers now

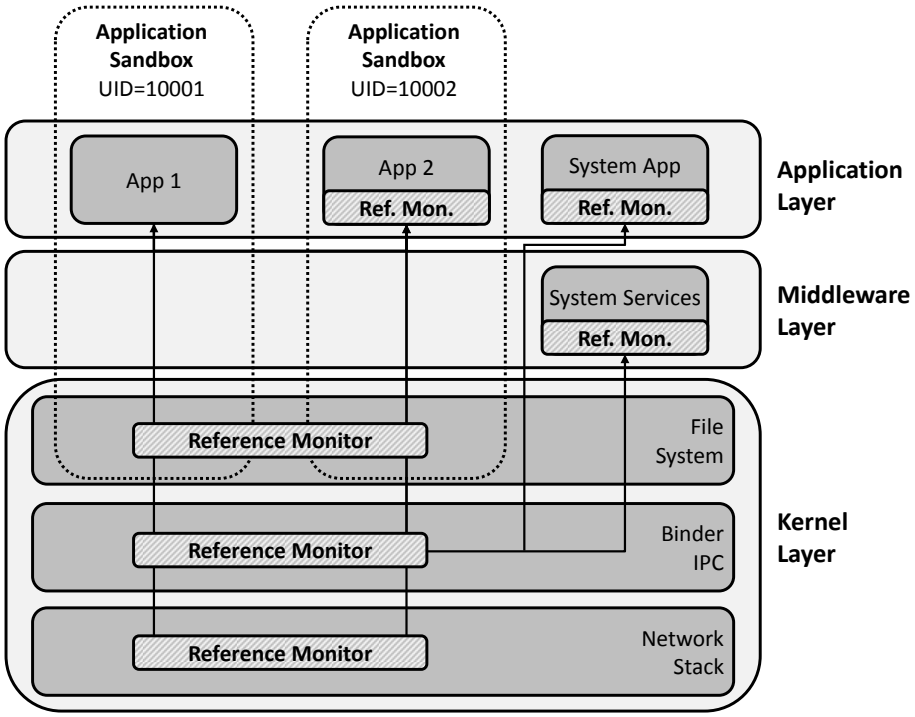


Figure 6: Android security architecture

have to handle permission revocation gracefully and expect users to not grant desired permissions.

Android permissions are generally enforced beyond the application process. Reference monitors in the Android operating system, most prominently in Android's system Services (e.g., ActivityManagerService and PackageManagerService) and preinstalled system applications, enforce these permissions during Binder IPC between applications at runtime (see Figure 6). Several permissions protecting low-level services, such as network and Bluetooth connectivity or access to external storage (e.g., removable Micro SD flash memory), are enforced by the Android Linux kernel. To this end, Android maps the corresponding permissions, such as the `INTERNET` permission, to Linux Group IDs (GIDs), which are linked to the application UID at installation time. Reference monitors within the filesystem and networking subsystems of the kernel at runtime check whether an application holds the corresponding permission by evaluating the GIDs of the process.

Finally, app developers can declare new permissions to protect their own application components. These permissions are declared in the manifest file and associated with the application components they are supposed to protect. App developers can further integrate reference monitors programmatically into their application code to enforce access control decisions dynamically at runtime on a sub-component level - for example to protect individual data fields within a ContentProvider.

3

ATTACKS AN DEFENSES

Smartphones and tablets process and store a vast amount of privacy- and security-sensitive data and provide constant access to the Internet and the telephony network. Malware abusing vulnerabilities in the operating system can thus directly or indirectly generate monetary revenue for malware authors, for example by exfiltrating valuable user data, performing ad fraud, or by establishing calls or sending text messages to premium rate services [190]. Considering that Android currently is the most widely used operating system for smartphones and tablets [130] it is not surprising that the number of attacks on Android has been rising steadily throughout its lifetime [160].

To mitigate such attacks both academia and commercial security solution vendors have proposed a variety of tools and use-case specific security extensions for Android. Many of these security extensions augment Android's default permission-based access control architecture, as we will show in this chapter. To facilitate the development of such security extensions without the hassle of modifying the operating system this dissertation seeks to promote extensible system-centric access control architectures. In the following, we will derive requirements for such architectures by first introducing a generic adversary model (see Section 3.1). We proceed to discuss important attack vectors available to the adversary to deploy malicious code (see Section 3.2) and to escalate its privileges (see Section 3.3). In Section 3.4, we will discuss mechanisms proposed by related work which aim to detect, analyze and mitigate these attacks. We conclude this chapter in Section 3.5 by deriving generic requirements for extensible access control architectures.

3.1 ADVERSARY MODEL AND SCOPE

This section describes the adversary model adopted within this dissertation. We first determine adversary goals and capabilities, and then define a set of reasonable assumptions which apply to all system-centric security architectures described within this dissertation.

Adversary Goals and Capabilities. In general, the goal of the adversary is to gain unauthorized access to security- and privacy-sensitive resources stored on or accessed by a smart mobile device. Such information comprises sensitive data stored within volatile or non-volatile memory, or exposed by application and operating system interfaces. Since this dissertation concerns extensible access control architectures for the Android operating system our adversary model focuses on *local adversaries*, which are able to deploy and execute arbitrary code on the Android operating system executed on the application processor. Unless otherwise stated, we do not consider remote adversaries using vulnerabilities in the network stack or protocols. We place no restrictions on the type of code the adversary can deploy on the target device and allow managed bytecode as well as

native and obfuscated code. Attack vectors to deploy malicious code on a target device are discussed in the following Section 3.2.

Assumptions. We assume that the adversary is unable to compromise the trusted computing bases (TCBs) of the security architectures presented within this dissertation. This assumption is reasonable, since by definition, compromise of the TCB will allow the adversary to circumvent security mechanisms implemented within the TCB. In general, the TCBs for the system-centric security architectures discussed within this dissertation consist of the boot loader (chain), operating system kernel and middleware layer as well as predeployed system applications. Further, we do not consider program code targeting the baseband processor, trusted execution environment or other peripherals which have direct memory access (DMA) to main memory. Such attacks would potentially allow an adversary to modify arbitrary memory structures under control of the Android operating system and thus to undermine any security mechanisms implemented on this level. We further do not consider Denial of Service (DoS) attacks, such as applications draining computational resources or battery power, or applications performing a factory reset of the device if authorized to do so by Android's security architecture. The mitigation of such DoS attacks is a problem well beyond the scope of this dissertation.

3.2 ATTACK VECTORS

Based on the capabilities of the adversary we will now systematically analyze different classes of attacks targeting privacy- and security-sensitive assets stored on or accessed by Android-based smart mobile devices. These attacks are based on the assumption that the adversary has deployed malicious code onto the target device. Related work has identified multiple attack vectors to deploy malicious code [304]. In general, these attack vectors can be categorized into two distinct classes, which we describe in the following.

3.2.1 *Active Deployment*

We consider an attack to use *active deployment* when it requires user interaction to install malicious code on his device. Applying this class of attacks to the Android operating system means that the user has to actively install an application package. He thus has to acknowledge the permissions requested by the application. To perform an active attack the adversary thus has to convince the user to install his malicious application. A number of weaknesses in Android's security architecture facilitate this process.

General Weaknesses of Android's Permission System. Related work has scrutinized Android's permission system and suggests that prominent issues are a limited understanding of risks associated with permissions [79] and a general overburdening of users with low-risk warnings. These problems cause users to grant permissions rather liberally [80], a behavior which is amplified by the fact that app developers routinely request more permissions than actually required by their applications [78]. It should be noted that overprivileged applications also violate the principle of least privilege adopted by Android's security architecture.

Another important limitation of Android's permission system stems from the fact that permissions are granted on the granularity of the Linux User ID (UID) assigned to applications at installation time. Accordingly, all components of an application are subject to the same permissions, regardless of the individual code components, their origin and their purpose. For example, consider a mail client, which legitimately needs access to the contacts ContentProvider as well as Internet connectivity to fulfill its purpose. However, at the same time this email client could silently upload all contacts data to a remote server under control by the malicious app developer. This lack of contextual integrity [281] is for example exploited by ad libraries, which developers integrate into free applications to retrieve and display advertisement. Related work has shown that such libraries routinely abuse the host app's permissions to silently access privacy- and security sensitive data on the device [241, 105].

Furthermore, permissions do not provide adequate fine-grained protection of privacy-sensitive data. Consider non-malicious but privacy-intrusive applications, which legitimately require access to certain records stored in the contacts ContentProvider. For example, it has been shown that popular social networking applications, such as "WhatsApp" and "Path", access a significant amount of contacts information without allowing the user to filter unwanted contacts, such as enterprise contacts [22, 76].

Finally, Android allows app developers to specify that his own applications should share a sandbox. This is realized using Android's *sharedUserid* feature [102], which assigns the same Linux UID to multiple applications signed with the same private key. Since technically permissions are assigned to Linux UIDs applications sharing the same UID share their home directories and their permissions. However, the user is not explicitly notified about this transfer of privileges.

No Central Deployment Authority. Android employs an open software deployment model and does not restrict users to Google's own "Play Store" app market. After activating a specific option in Android's system settings, users can install applications from arbitrary sources, such as third-party app markets or web sites. This process circumvents the (limited) app vetting mechanisms applied in Google Play Store [174] and impedes the central deployment of security updates and revocation of malicious applications [19].

Further, as described in Section 2.3.2 Android does employ mandatory code signing, but no central public key infrastructure is enforced upon developers. App signatures are only used to enforce a same-origin policy on app updates, and app developers generally use self-signed certificates for their applications. Accordingly, there is no reliable mechanism in place which prevents malware developers from integrating malicious functionality into existing application packages after stripping out the original certificate. This process is known as "application repackaging", and related work has shown that it is widely used to integrate malicious code into third-party applications, which are then distributed via third-party app markets or websites [88, 302, 301, 306, 300, 86]. Similarly, this lack of a central code signing authority facilitates software piracy.

3.2.2 *Passive Deployment*

In contrast to the previously described *active* deployment mechanisms *passive* deployment allows the adversary to install malicious software silently without any user interaction.

Software Vulnerabilities. The adversary can abuse software vulnerabilities in the Android OS and installed applications to deploy and execute code remotely. Related work has demonstrated that applications which dynamically load code fragments at runtime from remote servers can be abused to inject malicious code, which can then abuse the host application's permissions to access privacy- and security-sensitive data [189, 75, 73, 138, 156]. Furthermore, adversaries can exploit vulnerabilities in highly privileged software components, such as vendor-specific preinstalled system applications [173, 280, 163, 164], to deploy and execute code silently.

Temporary Physical Access. Finally, another approach to deploy software on a target user's device is to gain temporary physical access while the device is unlocked and to manually install malicious applications without the user noticing.

3.3 PRIVILEGE ESCALATION

Once the adversary has deployed malicious code onto a target user's device the deployed code is limited in its actions by Android's application sandbox and permission system. To access protected resources without authorization the adversary can exploit weaknesses in Android's security model. This process is called *privilege escalation*, and we will describe different variants in the following.

3.3.1 *Application-Layer Privilege Escalation*

Android's model of communicating software components executed in isolated least privilege application sandboxes has been shown to be susceptible to application-layer privilege escalation attacks [52]. Two distinct types of such attacks can be distinguished.

Confused Deputy Attacks. A confused deputy is a piece of software which is authorized to access protected resources and inadvertently exposes these resources to unprivileged code [115]. A malicious application can abuse the confused deputy to gain unauthorized access to protected resources. A wide variety of confused deputy vulnerabilities on Android's middleware layer, within system applications as well as third-party applications have been identified by related work. These vulnerabilities allow unauthorized applications to access protected interfaces for system settings [81], to connect to web servers [152], to access user credentials [74], to send SMS [52] or even to establish phone calls [68] without holding appropriate permissions.

Collusion Attacks. In a collusion attack an adversary deploys multiple cooperating applications on a target device. These applications by themselves appear benign and individually do not hold potentially dangerous combinations of permissions. At runtime however these applications coordinate their behavior via overt or covert communication

channels towards a common goal, such as accessing and exfiltrating privacy-sensitive data. In the context of the Android operating system, an overt channel is a purpose-built inter-process communication channel, such as Binder or Linux IPC. In contrast, any high- or low bitrate inter-process communication channel that has not been specifically designed for direct communication between applications constitutes a covert channel – for example, synchronized reading of and writing to Android system settings, log files or data stored within shared ContentProviders [32, 158]. Related work has demonstrated sophisticated collusion attacks which target credit card payments via smartphones and use covert channels for communication between colluding applications [218].

3.3.2 *Operating System Level Privilege Escalation*

Besides executing application-layer privilege escalation attacks malicious applications can target privileged and vulnerable operating system processes on Android's middleware and kernel layer.

Privileged Processes. Android executes a set of highly privileged processes on the middleware layer. By abusing security vulnerabilities in these processes the adversary can extend his privileges. Naturally, processes executed with administrative "root" privileges are of particular interest. Corresponding exploits are commonly known as "root exploits". For example, in Android 4.4.3 Google fixed an input validation vulnerability in the storage volume management daemon, denoted *vold*. Abusing this vulnerability allowed arbitrary processes to execute code with root privileges on an Android device [188]. Further examples are input validation vulnerabilities found in important Java classes, which ultimately allowed applications to gain administrative root privileges by targeting Android's SystemServer located at the middleware layer [123, 186].

Further valuable targets for adversaries are Android system components holding system-level permissions (see Section 2.3.2), which has recently been demonstrated by a set of vulnerabilities in Android's *mediaserver* component. Exploiting these vulnerabilities lead to arbitrary code execution privileges with system-level permissions [65].

Critical vulnerabilities have also been found in additional software integrated by device vendors into custom distributions of the open-source Android operating system. Using static program analysis of vendor-specific Android operating system images Wu et al. identified overprivileged and vulnerable preinstalled applications [285]. Similarly, Aafer et al. [2] used static taint analysis to detect privilege escalation vulnerabilities and information leaks introduced by the device vendor while customizing the Android operating system middleware for specific devices.

Linux Kernel. The operating system kernel is executed in a higher privileged CPU mode than userspace code, such as applications or privileged system processes. It is therefore a particularly interesting target for adversaries, since kernel-layer privileges allow the adversary to circumvent any security mechanisms implemented within kernel- and userspace. Note that this includes the kernel-layer SELinux/SEAndroid [153, 232] mandatory access control frameworks.

Since Android's initial release a number of vulnerabilities leading to arbitrary code execution privileges at the kernel layer have been discovered and fixed. For example, CVE-2014-3153 [256] describes a vulnerability in the Linux kernel's syscall interface which can be used by malicious apps to gain kernel-layer code execution privileges. This vulnerability was used by the popular *towelroot* exploit [125] and bypasses even the SELinux mandatory access control mechanisms deployed in recent Android versions.

Finally, vendor-specific modifications of the stock Android Linux kernel have been shown to increase its attack surface. Zhou et al. used dynamic program analysis to demonstrate that vulnerable vendor-specific device drivers in the Linux kernel pose a significant threat [303].

Higher-privileged CPU Modes. While out of scope of this dissertation, it should be noted that current CPU architectures used on Android-based smart mobile devices provide even higher privileged CPU modes designed for virtualization or the implementation of trusted execution environments. For example, recent ARM CPUs feature a *hyp* mode designed for running hypervisors [270]. Vulnerabilities in a corresponding hypervisor can potentially lead to compromise of the virtualized operating systems running in lower privileged CPU modes. Furthermore, vulnerable code running in CPU modes designed for implementing Trusted Execution Environments, such as the *secure* operation mode on ARM TrustZone capable SoCs [7], are valuable targets for attackers.

3.3.3 *Sensory Malware*

Smart mobile devices feature a significant amount of sensors observing the environment. Among those sensors are microphones and cameras, geolocation, climate and acceleration sensors. Related work has shown a variety of attacks abusing unrestricted sensor access, for example to derive sensitive user input, such as passwords, using the acceleration sensor [37, 290, 183, 159] or camera and microphone [228], or to recover credit card numbers from recorded speech samples [218].

3.3.4 *System and Application Updates*

Related work has identified that Android's system update process contains several vulnerabilities which allow applications to escalate their privileges [286]. These *pileup* attacks allow malicious applications to preemptively request undefined permissions or shared UIDs which will be introduced by subsequent Android versions. By abusing these vulnerabilities malicious apps can for example gain access to *SystemOrSignature* level permissions which are reserved for system applications (see Section 2.3.2) or gain complete access to system application sandboxes. More recent work discovered *data residue* vulnerabilities in Android's application uninstallation process which can cause leakage of app-specific privacy- and security sensitive data, such as credentials for online services, to malicious apps [297].

3.3.5 *User Interface Confusion*

Weaknesses in Android's Activity management allow malicious applications to inject themselves into user interface workflows and to trap the user in full screen applications without his knowledge [206, 25]. Attacks abusing these vulnerabilities enable malicious applications to extract sensitive user data, such as login credentials for mobile payment services or social networks.

3.4 THREAT MITIGATION

The amount of threats and vulnerabilities targeting smart mobile devices in general and the Android operating system in particular, as well as the limited effectiveness of anti-malware products for these devices [202, 304] has motivated academic and industrial research to develop novel security mechanisms. In the following, we will provide an overview on corresponding related work, giving special attention to enhancements of Android's access control mechanisms.

3.4.1 *Static Program Analysis*

Static analysis tools employ reverse-engineering techniques to reason about program behavior without executing a target program. In the Android context, they operate on application code, metadata (e.g., the application manifest containing all defined and requested permissions) as well as bundled resources, such as text files and images. First and foremost, related work has embraced static analysis to analyze Android applications for potential security- and privacy violations, undesired leakage of sensitive data, unprotected Binder IPC interfaces as well as malicious program behavior [20, 187, 67, 175, 44, 78, 155, 84, 9, 292, 87, 104, 106, 82, 305, 295, 279, 142, 8, 11, 177, 176, 141, 83, 41, 157]. Hu et al. demonstrate how to use static program analysis to verify the integrity of third-party code libraries [87]. While the detection of repackaged applications [88] is a prominent use-case for static and hybrid program analysis [55, 302, 301, 42, 56, 47, 48, 111, 150, 200], which combines static and dynamic analysis techniques, more recent approaches also consider similarities in application resources, such as text strings or user interface components, with promising results [294, 137, 43, 223, 243, 235, 93].

The Android sourcecode itself is an interesting target for static analysis as well. As part of the Stowaway project [78] Felt et al. generated a mapping between Android APIs and corresponding permissions by manually inspecting the Android sourcecode. In contrast, the PScout [10] project derived a more precise mapping using automated static program analysis. More generally, SuSI [198] identifies privacy-critical data sources and sinks in the Android operating system using a machine-learning approach. EdgeMiner [39] focuses on security aspects of callback-driven components in Android's middleware layer. The recently proposed Kratos [224] architecture uses static code analysis to detect inconsistencies in permission enforcement between different Android APIs which provide access to the same functionality.

In the Android context the proposed static analysis frameworks face several general challenges. First, static analysis by design only considers program code at the time of analysis. Applications modifying their code at runtime, for example by downloading additional DEX bytecode, can conceptually not adequately be addressed. Second, most static program analysis tools only operate on DEX bytecode and do not consider native code, and are thus unable to completely analyze a large subset[1] of Android applications.

3.4.2 *Dynamic Program Analysis*

Approaches using dynamic program analysis augment static analysis by observing program execution in instrumented environments. In the Android context, common techniques adopted by these tools are hooking of security- and privacy-sensitive application-, middleware- and kernel-layer components [222, 244, 306], system call tracing [27, 210, 35] and dynamic taint analysis [66, 299]. Further, several works use virtual machine introspection on the level of the Dalvik Virtual Machine [148, 151], application process [237], the QEMU-based Android emulator [248, 291] or a combination of these levels [202]. Some approaches additionally explicitly integrate network traffic analysis [272, 151]. Finally, related work has shown that the combination of symbolic application execution and static taint analysis is a promising approach for the detection of privacy leaks in Android applications [293].

To improve scalability related work has proposed distributed architectures for dynamic behavior analysis. Paranoid Android [191] demonstrates how dynamic program analysis can be offloaded to external servers by mirroring program execution traces to emulated mobile devices in the cloud. On the other hand, AirMid [167] uses a network intrusion detection system (NIDS) to identify malicious network traffic of smart mobile devices. The NIDS component communicates with an on-device agent deployed as part of the operating system kernel to neuter applications generating malicious network traffic.

Similar to the proposed static analysis tools dynamic program analysis also faces several challenges. A general concern is that these approaches are prone to logic bombs, where programs attempt to detect instrumented environments or delay their malicious behavior to avoid detection while being analyzed [201, 271, 139]. Further, while in the Android context dynamic taint analysis has been shown to be a promising approach for tracing explicit information flows in Java-based applications [66, 124], current designs do not adequately handle native code or implicit information flows, and are susceptible to side-channel attacks [12].

3.4.3 *Root Exploit Mitigation*

Fedler et al. discuss the possibility of introducing more fine-grained system-centric control on native code execution on Android [77]. They propose the adoption of MAC and DAC policies to control which third-party applications may load and execute third-party native

[1] While Google does not publish official statistics, recent large-scale studies indicate that up to 37% of all Android applications use native code [4].

code at runtime. PREC [121] aims to contain root exploits used by malicious Android applications. The basic idea is to force applications to adhere to a pregenerated specification, which describes system calls executed by the application's native code components. At runtime, PREC compares the behavior of native code components with this model and either completely shuts down or exponentially slows down suspicious app components. While the authors demonstrate that this exponential slowdown can prevent certain root exploits relying on the execution of a high amount of syscalls, it is debatable whether or not this approach can be generalized.

3.4.4 *Fine-grained Privilege Separation*

Several approaches attempt to integrate more fine-grained privilege separation into the Android operating system. NativeGuard [242] isolates third-party native code into separate unprivileged processes subject to strict access control. However, it still allows all calls from native code to an application's higher-privileged Java bytecode, which arguably defeats the purpose of native code isolation. Furthermore, a recent large-scale study on native code usage in real-world Android applications [4] has shown that depriving native code in Android apps of all privileges will cause a considerable amount of existing applications to malfunction.

Compac [276] aims to enforce access control rules on Android components on the subprocess level. It introduces additional component permissions, which it enforces by runtime inspection of the call stack. Compac is however susceptible to attacks using native code or dynamic code execution, and given that users are often overburdened with Android's existing permission-based access control model (see Section 3.2.1) it is debatable whether or not component permission will improve the current situation. Nonetheless, the more recent FlexDroid [220] architecture similarly imposes a separate set of permissions on 3rd-party code libraries integrated into Android applications. However, their approach, denoted *inter-process stack inspection*, uses hardware fault isolation and is resilient to attacks using native code or dynamic code execution.

Addroid [185], AdSplit [225] and AFrame [296] introduce privilege separation targeting advertisement libraries. These approaches isolate ad library code from the main application process and execute them with a limited set of permissions, which effectively mitigates the risks of ad libraries abusing the host app's permissions. LayerCake [208] is a more generic approach which allows developers to securely embed Activities of other applications (e.g., advertisement clients) or the operating system (e.g., an isolated web viewer Activity) within the user interfaces of their own apps. Privilege separation is achieved by running the embedded Activities in separate processes, which are isolated from the original application.

3.4.5 *System and Application Updates*

Google provides security updates for AOSP and the Nexus series of Android devices. However, other vendors often do not port these security updates to their device-specific

versions of the Android OS. PatchDroid [165] addresses this concern by patching relevant code in volatile memory on devices for which no official security updates are available.

3.4.6 *System-Centric Access Control Refinement*

Multi-layer Access Control Frameworks. Android Security Framework (ASF) [14] provides a programmable interface to implement security modules on Android. Similar to the architectures presented in this dissertation it promotes the need for a programmable interface for developing reference monitors. It provides the required authorization hooks in the Android operating system and supports mediation in third-party applications. It further enables sub-process level enforcement using inlined reference monitors [70] (IRMs, see Section 3.4.7). It should be noted that ASF has been developed concurrently and independently of our work and achieves similar goals. However, in ASF the security module developer is generally completely trusted. Thus, a vulnerable ASF module can undermine secrecy and integrity of the Android operating system and all installed applications.

Kernel-Layer Mandatory Access Control. Modern operating systems and security extensions for smart mobile devices employ kernel-layer mandatory access control mechanisms to harden application sandboxes. As described in Section 2.3.2 Android in particular adopts the SELinux [153] kernel-layer mandatory access control framework. While Shabtai et al. were the first to propose the adoption of SELinux for Android [221] Google decided to merge the more recently developed SEAndroid [232] variant to strengthen application sandboxes. SEAndroid uses domain and type enforcement policies to restrict processes to operations they fundamentally require, as we describe in more detail in Section 4.1. Vendor-specific SEAndroid access control policies have been scrutinized by related work, which identified problematic patterns motivating better tools for SEAndroid policy analysis and development [207]. One particularly interesting approach is the use of machine learning techniques to analyze SEAndroid audit logs on a massive scale [274] to improve the access control policy.

Notably, kernel-layer mandatory access control mechanisms have been embraced by other operating systems for smart mobile devices as well. SELinux has been prototypically deployed on the OpenMoko [166] and LiMo (Linux Mobile) platform [298]. In contrast, the security architecture of Tizen [255] relies on SMACK [216], while Apple iOS adopts the TrustedBSD framework [277].

Mitigating Application-Layer Privilege Escalation Attacks. Confused deputy and collusion attacks operate on the application layer and cannot adequately be addressed by kernel-level access control mechanisms alone. XManDroid [32] uses access control hooks deployed in sensitive components on both the middleware and kernel layers of the operating system to generate a global view of application interaction. Using these hooks XManDroid tracks inter-process communication over both overt and covert communication channels and generates a graph-based representation. Based on this component interaction graph XManDroid enforces access control policies targeting both confused deputy as well as collusion attacks.

Further, related work has discussed the mitigation of confused deputy and collusion attacks based on IPC call chain verification. These approaches use similar mechanisms to trace IPC call chains across process boundaries at runtime [13, 81, 58]. The generated call chain information can then be used to restrict access to sensitive resources in case untrusted or underprivileged applications appear on the call chain.

Permission Refinements. APEX [169] integrates revocable permissions as well as context-dependent permission constraints into the Android operating system. It is deployed by extending Android's permission logic at the middleware level with additional access control hooks. Permission assignment can be changed at installation time by the user. However, enforcement is typically not graceful, which may cause applications accessing protected APIs while their permissions are revoked to crash [144]. The root cause is that Android's app development model until version 6.0 did not require the app developer to check whether or not an app holds a requested permission at runtime. TISSA [307] and MockDroid [24] allow users to dynamically revoke privacy-critical permissions gracefully, which means that instead of denying applications access to privacy- and security critical resources their access control hooks return bogus or anonymized data. Kirin [69] uses an alternative approach and checks for dangerous combinations of permissions at installation time. In case a dangerous combination of permissions is detected it simply denies the installation of the application.

DeepDroid [275] enforces access control policies on Android applications by dynamically hooking system services and tracing system calls performed by applications. While dynamically hooking system services using the kernel's debugging interfaces, such as the ptrace mechanism, does not require replacing the vendor-specific Android operating system, DeepDroid still needs root privileges, a feature which is not available on regular Android distributions without compromising system integrity.

Application Hardening. Saint [180] allows app developers to integrate sophisticated access control policies into their applications. A system-centric access control framework evaluates these policies first at installation time to block the installation of apps which do not conform to all deployed access control policies, and later dynamically at runtime during inter-process communication. Saint's access control policies can be based on application identity and metadata as well as device runtime configuration.

AppSealer [295] aims to mitigate the threat of application components inadvertently exported to other applications via Binder IPC interfaces. Such vulnerabilities in applications can constitute confused deputy attacks in case they leak capabilities to other apps. AppSealer uses static taint analysis to identify such vulnerabilities in Android application packages and binary rewriting to generate corresponding security patches without access to the application sourcecode.

Similar to DeepDroid [275] the FireDroid architecture [210] uses ptrace to instrument the zygote process and to apply security policies to all installed applications. While it does not require modifications to the Android distribution or application sourcecode the target device needs to be rooted.

Information Flow Control. AppFence [124] builds on TaintDroid's dynamic taint analysis engine [66] to enforce access control rules targeting privacy-intrusive apps. It pro-

vides two enforcement mechanisms: First, it replaces privacy- and security-sensitive data with mock data at runtime if desired, which is comparable to TISSA's [307] and MockDroid's [24] access control enforcement mechanisms. Second, it modifies Android's network stack to evaluate taint information at runtime and prevents tainted privacy-sensitive data from being exfiltrated via network sockets.

Aquifer [168] introduces user interface workflow policies to Android. Using a system-centric access control architecture operating on both kernel- and middleware layer Acquifer enforces application interaction and information flow policies on users, for example to impose access and export restrictions on confidential email attachments downloaded via the email client.

Digital Rights Management. Porscha [179] enforces DRM policies on sensitive data objects in transit and on Android-based devices. Data in transit is protected using identity-based encryption [28]. Data stored on the device is protected from illegitimate access using additional access control mechanisms deployed in modified Android middleware components and system applications.

Security Domain Isolation. Several approaches aim to isolate different security domains on smart mobile devices, such as the enterprise and private domain. PINPOINT [203] introduces namespaces to Android's middleware layer to enable user-controlled isolation of applications belonging to different security domains on the application and middleware layer. MOSES [209] provides security domain isolation on the application-, middleware and kernel layer based on information flow control using TaintDroid [66], which can however not adequately address third-party native code.

To provide *complete* mediation on *all* layers of the operating system related work has discussed three basic techniques, which are depicted in Figure 7. Several approaches use virtualization, either based on bare-metal hypervisors [107], Linux-based paravirtualization [18] or the L4 Microkernel [146], to execute complete operating system stacks in isolated environments. Alternatively, kernel-layer compartments [5, 284, 287] provide similar isolation guarantees by executing separate Android middleware- and application-layer environments using the kernel's namespace mechanisms. Finally, a set of extensions to Android's operating system middleware and kernel layer implement security domain isolation using mandatory access control [33, 214, 133].

Despite their comparatively small TCB footprint both virtualization and kernel-layer compartments are rarely used in practice on smart mobile devices today. The main reason is that they require the duplication of large parts of the software stack. In contrast, approaches integrating sophisticated mandatory access control architectures into Android's middleware- and kernel layer have proven to be more practical and are available as commercial solutions today. The TrustDroid architecture [33], which laid the foundation for the commercial BizzTrust solution (see Section 6.3), as well as the technologically comparable Samsung KNOX architecture [214] deployed on select Samsung smartphones use kernel- and middleware-layer mandatory access control to isolate applications of different security domains. This trend towards system-centric access control is further acknowledged by Google's introduction of "Android for Work" in Android 5.0 [133], which uses similar mechanisms to isolate applications of different security domains.

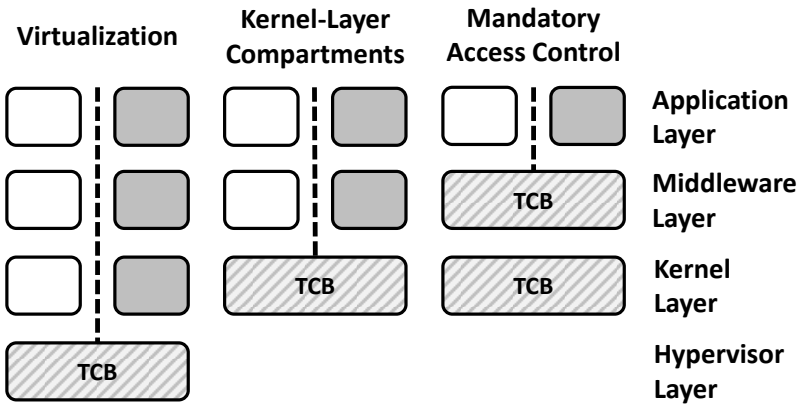


Figure 7: Different approaches for security domain isolation

3.4.7 *Application-Layer Access Control Refinement*

To avoid modification of operating system components related work has proposed the use of binary rewriting to enforce more strict access control rules on applications. In contrast to system-centric access control frameworks these approaches do not require changes to the underlying operating system.

Dr. Android and Mr. Hide [136] proposes a combination of binary rewriting of Android applications and application-layer deputies. Calls to sensitive APIs are replaced by calls to a deputy application, which is highly privileged and enforces access control decisions. The original Android app is then stripped of all legacy permissions. Alternatively, app developers can directly target the deputy API instead of Android's own services, which is a viable alternative for applications targeting multiple operating systems (see Section 6.4). In contrast, inlined reference monitors (IRMs) [70, 16, 289, 54, 53, 199] integrate additional access control mechanisms into the application process itself. However, in IRM-based solutions the reference monitor is not more privileged or even logically separated from the application process it is designed to enforce policies on. Therefore, these approaches are susceptible to attacks using native code to disable the reference monitor at runtime [113]. Finally, it should be noted that binary rewriting generally breaks Android's app update process due to the employed same origin policy (see Section 2.3.2).

Boxify [15] uses a similar approach to Dr. Android and Mr. Hide but does not require any changes to the application binary. Instead, Boxify uses Android's *isolatedProcess* feature, which spawns permission-deprived sandboxes for individual processes to execute Android applications within an isolated environment. By redirecting calls to sensitive interfaces, such as system calls and Binder IPC transactions, via an application-layer deputy Boxify effectively isolates applications without introducing changes to the underlying operating system. The independently and simultaneously developed NJAS architecture [26] achieves similar goals using system call interposition to start a target application under control of a monitoring app.

3.5 REQUIREMENTS FOR EXTENSIBLE ACCESS CONTROL ARCHITECTURES

Based on our previously described adversary model, attacks on Android's security architecture as well as the proposed extensions to Android's access control architecture we will now derive requirements for extensible system-centric access control architectures.

3.5.1 *Observations*

While analyzing the proposed *system-centric* Android security extensions we observed that many of them primarily focus on extended access control for application communication channels, such as Binder and Linux IPC, network sockets or the file system. Accordingly, while their use cases are diverse, these extensions integrate similar hooks into Android's default access control mechanisms. However, the proposed extensions mostly address domain-specific issues and are not generic.

Table 3 identifies relevant prior work on system-centric process-level access control and classifies it by authorization hook semantics. Since this dissertation promotes extensible system-centric access control we operate under the assumption that the operating system will be modified *once* to integrate the access control architecture. Accordingly, we focus on those proposals which must be integrated at build time. Approaches operating on the sub-process level, which is the designated security boundary for Android applications, are out of scope of our analysis. Such approaches include those based on inlined reference monitors, runtime memory instrumentation as well as dynamic taint analysis. We further do not consider approaches that rely on debugging interfaces, such as ptrace, since their main contribution is merely an alternative to the modification of central operating system components to extend Android security.

Nearly all of the corresponding proposals hook into Android's Binder ICC mechanism and corresponding enforcement points in ActivityManagerService. The PackageManagerService (PMS) is also frequently instrumented to modify application permissions. Permissions are also occasionally customized by modifying the interfaces to device sensors and system ContentProviders containing privacy sensitive information (e.g., contacts data). Several proposals also require authorization hooks for file and network access, which are enforced in the Linux kernel.

The table also denotes two areas that are nonstandard for OS reference monitors. The first hook semantics is the use of fake data. That is, instead of simply allowing or denying a protected operation, the hook must modify the value that is returned. This third option is often essential for protecting user privacy while maintaining usability via graceful revocation. For example, the geographic coordinates of the North Pole, or maybe coarse city coordinates can be substituted for the devices actual location. Replacing unique identifiers (e.g. IMEI or IMSI) to combat advertising tracking is a further example. The second interesting hook semantics is the inclusion of third-party hooks. That is, a third-party application wishes the OS reference monitor to help enforce its security goals.

Table 3: Classification of authorization hook semantics required by system-centric Android security enhancements

System	Android ICC	Package Manager	Sensors / Phone Info	Fake Data	System Content Providers	File Access	Network Access	Third Party Extension
MockDroid [24]		✓	✓	✓	✓		✓	
XManDroid [32]	✓	✓	✓			✓	✓	
TrustDroid [33]	✓	✓			✓	✓	✓	
Porscha [179]	✓				✓	✓	✓	
CRePE [45]	✓		✓					
Quire [58]	✓	✓						
Scippa [13]	✓							
TaintDroid [66]	✓		✓			✓	✓	
Kirin [69]		✓						
IPC Inspection [81]	✓	✓						
AppFence [124]	✓	✓	✓	✓	✓	✓	✓	
Aquifer [168]	✓					✓	✓	
APEX [169]	✓	✓	✓					
Saint [180]	✓	✓						✓
SEAndroid [232]	✓	✓				✓	✓	
TISSA [307]			✓	✓	✓			
PINPOINT [203]	✓		✓					

3.5.2 *Requirement Analysis*

Based on the previously described Android operating system architecture, our adversary model and our observations regarding system-centric security extensions we will now derive a set of requirements for extensible access control architectures.

1. **Generic Authorization Expressibility.** Extensible access control frameworks should enable the implementation of both prior and future security enhancements for Android. This requirement mandates extended access control mechanisms on all operating system layers and support for context-aware access control. Generic expressibility also covers the ability to replace arbitrary data values [24, 307], which enables graceful enforcement without legacy applications crashing whenever operations are unexpectedly denied.

2. **Preservation of Existing Security Guarantees.** Android provides sandboxing guarantees to application providers. Allowing third-parties to extend Android's security framework potentially breaks those guarantees. Therefore, an extensible access control architecture should by default only make enforcement more restrictive (e.g., fewer permissions or less file system access).

3. **Protection of Kernel Integrity.** As an explicit extension to Goal 2, we must maintain kernel integrity. Some existing security extensions require additional access

control mechanisms within the Linux kernel. We cannot provide the control over kernel-layer access control mechanisms to third-parties without some controls.

4. **Support for Multiple Stakeholders.** As described in Section 2.1.1 smart mobile devices operate in an ecosystem subject to the interests of multiple stakeholders, such as end users, device manufacturers, enterprise administrators and app store providers. The interests of these stakeholders do not necessarily align. Extensible access control architectures should thus be able to consider and consolidate access control rules deployed by multiple stakeholders. This aspect mandates a consolidation mechanism to handle conflicting rules. We note that while related work has shown that it is impossible to provide a generic solution for access control policy consolidation, use-case specific consolidation strategies are feasible [204, 126].

5. **Minimal Resource Overhead.** While a certain performance decrease is inevitable due to the additional mediation extensible access control architectures should minimize the impact on system resources.

In the following Sections 4 and 5 we will discuss two approaches to address these requirements. We will first introduce FlaskDroid [34], which prototypes extensible *policy-driven* access control on Android. We proceed to generalize this approach in the Android Security Modules (ASM) framework [119], which brings *programmable* access control to the Android operating system.

4

FINE-GRAINED AND EXTENSIBLE POLICY-DRIVEN ACCESS CONTROL

The goal of the FlaskDroid framework presented in this section is to enable the policy-driven instantiation of security extensions without the need to introduce further changes into Android's sourcecode. To achieve this goal, FlaskDroid introduces type enforcement into Android's kernel, middleware and application layer – a challenging task due to the different semantics of these layers. An efficient policy language allows the dynamic instantiation of many of the security mechanisms proposed by related work and discussed in the previous section. We present our design and instantiate FlaskDroid based on the SEAndroid project [232], which prototypes SELinux type enforcement [153] on Android's Linux kernel layer and has concurrently and independently of our work been integrated into mainline Android in version 4.3. We demonstrate the extensibility of our approach by instantiating a set of interesting use cases. Finally, we evaluate FlaskDroid's performance impact and security guarantees in an automated test suite.

Contribution. To summarize, our main contributions are as follows:

- **Type Enforcement on Android's Kernel, Middleware and Application Layer.** Our FlaskDroid framework [34] builds on the established SELinux/SEAndroid kernel-layer MAC framework [153, 232] and for the first time extends type enforcement to Android's middleware- and application layer.
- **Policy-based Interface for System-centric Access Control.** Our access control architecture and policy design bridges the semantic gap between kernel- and userspace data structures and access control mechanisms.
- **Instantiation of Use-Case Specific Security Solutions.** FlaskDroid for the first time demonstrates that it is possible to instantiate many existing security solutions by merely designing a corresponding access control policy.

The rest of this chapter proceeds as follows: Section 4.1 introduces necessary domain-specific background knowledge on type enforcement in general and SELinux/SEAndroid in particular. In Section 4.2 we present the design and implementation of our FlaskDroid architecture, followed by a discussion of practical use-cases we implemented using FlaskDroid in Section 4.3. We evaluate Flaskdroid's performance overhead and effectiveness in Section 4.4. Finally, Section 4.5 provides concluding remarks.

Remark. The results presented in this section were achieved in collaboration with Sven Bugiel and Ahmad-Reza Sadeghi. In addition, N. Asokan and Steven Smalley were involved in initial discussions of the high-level idea. FlaskDroid's design was generated in a joint effort by Sven Bugiel, the author of this dissertation and Ahmad-Reza Sadeghi. Sven Bugiel was responsible for central aspects of our implementation, while the author

contributed several important Userspace Object Managers, as well as communication interfaces between ContextProviders, native code components and the central SecurityServer. He further designed and implemented our modifications to Android's AIDL compiler. Performance and policy complexity evaluation are due to Sven Bugiel, while effectiveness was analyzed by the author of this dissertation. Further, the author was responsible for the use-cases "Phone Booth Mode" and "Privacy Enhanced Image Media Store", while Sven Bugiel contributed "Privacy Enhanced Operating System Components" and "App Developer Policies (Saint)".

4.1 BACKGROUND ON SELINUX TYPE ENFORCEMENT

In this section, we introduce additional background knowledge which is specific to the FlaskDroid architecture and augments Section 2.3.

4.1.1 *SELinux*

FlaskDroid is based on SELinux, which is a (primarily) kernel-level mandatory access control architecture derived from Flux Security Kernel (Flask) [236]. In the Flask architecture, access control decisions are decoupled from access control enforcement. In SELinux this concept is realized using a central SecurityServer deployed in the Linux kernel, which is responsible for the runtime management of access control rules defined in an access control policy. Various policy enforcement points, denoted Object Managers, place hooks into security-sensitive subsystems of the Linux kernel and enforce the access control decisions of the kernelspace SecurityServer.

SELinux supports a variety of access control models, such as type enforcement, role-based access control or multilevel security. Our FlaskDroid architecture uses type enforcement, a mechanism which describes access control rules based on *types* assigned to *subjects* (i.e. processes) and *objects* (e.g., specific files). Objects belong to *object classes*, which are used to define common *operations* on them (for example, *read* and *write* operations defined for the *socket* or *file* object classes). *Attributes* allow grouping of related types for convenience. An access control rule defines which operations a subject type is allowed to perform on a specific object based on the type and class of the object. Listing 1 abstractly describes an allow rule which grants a set of subject types T_{Sub} authorization to perform a set of operations O_C on a set of objects of type T_{Obj} and class C_{Obj}.

Listing 1: SELinux allow rule based on the SELinux policy language specification [109, 34].

```
1 allow T_Sub T_Obj : C_Obj O_C;
```

SELinux provides limited support for dynamic policies, where access control rules depend on the state of the system. In particular, *policy Booleans* (see Listing 2) describe conditions depending on which a rule is considered to be active. A common use for such policy Booleans is switching between SELinux *permissive* and *enforcing* mode: While in permissive mode SELinux merely writes policy violations into an audit log file, any opera-

tion not specifically allowed by an access control rule is prevented while SELinux operates in enforcing mode.

Listing 2: Example usage of policy Booleans in a SELinux access control policy. Only if the value of `allow_access` is `true` the rule is active [34].

```
1 bool allow_access=true;
2 if (allow_access) {
3   allow T_Sub T_Obj : C_Obj O_C;
4 }
5 else {
6   ...
7 }
```

While SELinux has its roots in microkernel security [236], where many security sensitive operations are performed in userspace, the primary use of SELinux today is access control enforcement within the Linux kernel. However, the SELinux architecture provides support for access control in userspace based on Userspace Object Managers (USOMs). These USOMs are policy enforcement points for data structures specific to userspace components. USOMs assign object types to specific objects they manage. For example, in the Android context the contacts ContentProvider would operate as an USOM by assigning types to the individual fields constituting a contact record, such as a contact's name or telephone number. USOMs enforce access control decisions according to an access control policy managed by a SecurityServer, which can either be implemented as a separate userspace component or as part of the existing kernelspace SecurityServer.

4.1.2 *Security Enhanced (SE) Android*

SEAndroid [232] is a port of SELinux [153] for the Android operating system. Its main purpose is to harden the operating system against privilege escalation attacks by confining privileged system processes into least privilege sandboxes and further to strengthen Android's default application sandboxes. SEAndroid takes peculiarities of the Android operating system into account which do not apply to standard Linux distributions. Most importantly, SEAndroid respects the Android application lifecycle model, where application processes are forked from the zygote process, and applies types to processes based on application package metadata. SEAndroid has been integrated into mainline Android since version 4.3 and is operating in full enforcement mode since Android 5.0 [101].

SEAndroid initially contained a set of Android-specific mandatory access control mechanisms for userspace components, such as mediation for Intents and ContentProviders, an install-time mandatory access control mechanism to enforce restrictions on application installation and a permission revocation mechanism [234]. These mechanisms have partially been obsoleted by recently introduced features in mainline Android (e.g., revocable permissions in Android 6.0 or the unofficial Intent firewall [40]) or abandoned.

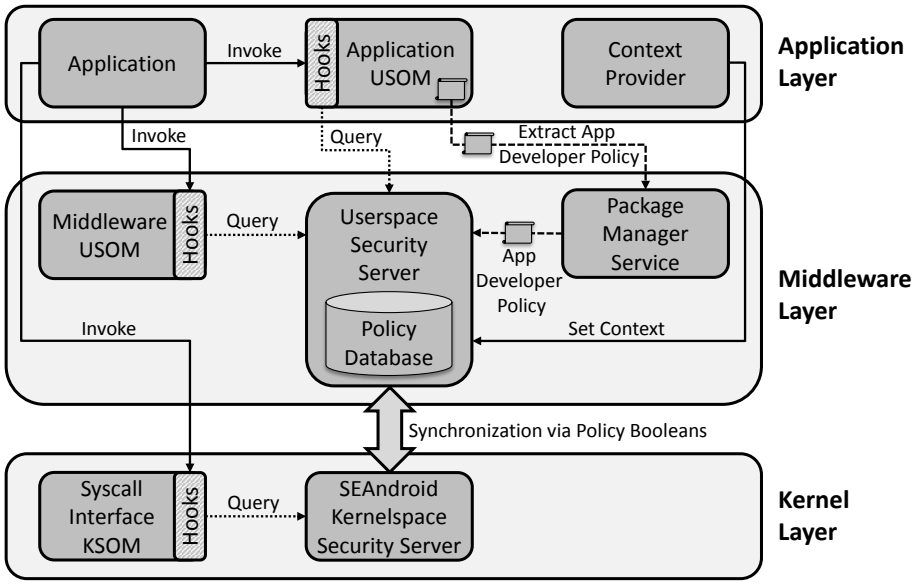


Figure 8: FlaskDroid framework architecture

4.2 FLASKDROID ARCHITECTURE

The basic idea of FlaskDroid is to adopt the Flask architecture [236] to Android by deploying Object Managers in privacy- and security-sensitive components on both the kernel- and middleware layer to augment Android's default access control mechanisms. These Object Managers assign types to the objects they manage. Kernelspace Object Managers (KSOMs) are responsible for enforcing access control decisions on low-level objects, such as files and sockets. In contrast, middleware- and application layer Userspace Object Managers (USOMs) control access to higher-level abstractions, ranging from Android Services and Intents to individual records stored in ContentProviders.

Whenever applications attempt to access privacy- or security-sensitive objects, our Object Managers query a SecurityServer for access control decisions. The SecurityServer evaluates the active security policy and instructs the Object Managers to allow or deny the applications' requests. This separation between Object Managers and SecurityServers decouples access control enforcement from access control decisions.

While recent Android versions employ SELinux type enforcement on the kernel layer to harden the operating system FlaskDroid extends type enforcement to the middleware- and application layer. It further provides a unified policy-based interface to define access control rules. To this end, our SELinux-based policy language introduces new features specific to Android's middleware and application-layer semantics and to address our requirements for supporting multiple stakeholders and context-awareness. Figure 8 provides an overview of our architecture, and we will explain the individual components in the following.

4.2.1 *Kernel-layer Type Enforcement*

Our kernel-layer access control mechanisms are provided by SEAndroid [232]. This design choice is driven by the following observations:

First, SEAndroid hardens the trusted computing base on the application and middleware layer against privilege escalation attacks. To do so, it confines both applications and highly privileged operating system services into isolated kernel-layer least privilege domains which only have access to resources they fundamentally require for correct operation.

Second, SEAndroid provides the necessary SecurityServer and Object Managers on the kernel layer. These components prevent applications from subverting our middleware-layer type enforcement architecture by accessing privacy- or security-sensitive resources via the operating system kernel. For example, applications could access records stored in a protected ContentProvider by directly reading the corresponding SQLite database file in case the discretionary access control rules are not configured correctly. Dynamically aligned middleware- and kernel layer policies ensure that FlaskDroid protects these resources on both the kernel- and middleware layer.

4.2.2 *Userspace Security Server*

Our architecture extends type enforcement to Android's middleware and application layer, which are both located in the userspace environment. Accordingly, our architecture mandates the presence of a policy decision point for managing access control policies concerning userspace types. In general, there are two distinct options to implement this functionality.

The first option is to reuse the SecurityServer deployed by SELinux/SEAndroid in kernelspace and to define all access control rules within the corresponding security policy. While straightforward, this approach shifts policy rules for data structures which semantically only apply to userspace objects into the kernel. It further introduces context switches whenever a Userspace Object Manager is invoked. Unsurprisingly, this approach is not being pursued by related work [261].

Another option is to introduce a separate Userspace Security Server, which is responsible for managing all access control rules enforced by Userspace Object Managers. In FlaskDroid, we chose to pursue this second option since it provides a clean separation of duties between user- and kernelspace. Upon boot, the Userspace Security Server loads all access control policies deployed by different stakeholders. Userspace Object Managers query the Userspace Security Server via inter-process communication (IPC) interfaces to mediate access control queries. Our Userspace Security Server resolves policy conflicts and communicates its decisions to the corresponding Userspace Object Managers. It further provides an Access Vector Cache, which caches access control decisions to improve performance.

4.2.3 *Userspace Object Managers*

As noted before, Userspace Object Managers assign types to the objects they manage and act as policy enforcement points for access control decisions yielded by the Userspace Se-

curity Server. We will now discuss important Userspace Object Managers implemented in our architecture.

PackageManagerService. Android applications are distributed as application packages, and PackageManagerService is responsible for managing these packages on the device. PackageManagerService upon application installation verifies application signatures and registers application metadata, such as the application's components and permissions, with the Android operating system. Finally, it assigns a Linux User ID (UID) to the application, which can later be used to identify the application during IPC. As described in Section 2.3.1 PackageManagerService performs a component lookup when an application invokes another application's component or an operating system component via an Intent.

In FlaskDroid, PackageManagerService operates as a Userspace Object Manager and assigns userspace types to application sandboxes. More specifically, PackageManagerService assigns types to the corresponding Linux UIDs depending on application metadata, such as the application package name, developer signature or an additional signature embedded into the application package which matches a public key deployed on the device (see Listing 3). System applications are assigned a consolidated kernel- and middleware type statically, while third-party apps are labeled at installation time.

As discussed before PackageManagerService is responsible for component lookup when applications interact via Binder IPC. Recall that PackageManagerService internally resolves Intent messages based on Intent metadata (see Section 2.3.1). As a Userspace Object Manager Android's PackageManagerService filters the list of candidate components according to policy decisions by the Userspace Security Server, which prevents an application from invoking another application's component unless the access control policy explicitly allows this operation.

Finally, our FlaskDroid architecture allows app developers to include access control policies concerning their own application components in their Android application packages. PackageManagerService extracts these policies at installation time and forwards them to the Userspace Security Server for mediation. This mechanism is required to address scenarios where multiple stakeholders are involved, and we discuss this aspect in more detail in Section 4.2.6.

ActivityManagerService. Android's ActivityManagerService is a central system component responsible for application Activity lifecycle management as well as Intent handling. Accordingly, as a Userspace Object Manager it handles Activity and Intent classes. Activities inherit the type assigned to the UID of the applications they belong to. Intents on the other hand are assigned a type dynamically at runtime based on Intent metadata, as shown in Listing 4. ActivityManagerService further enforces access control rules on Activity operations, such as starting and stopping Activities or moving them to the foreground. Finally, ActivityManagerService enforces access control on Broadcast Intents, which are Intents delivered to multiple applications' Broadcast Receivers. Before delivering these Broadcast Intents, our modifications to ActivityManagerService filter the list of candidate Broadcast Receivers according to the access control policy.

Listing 3: FlaskDroid policy excerpt describing application types. The class definition `app_c` defines operations applicable to applications. Applications are assigned the `app_unknown_t` type unless otherwise specified [34].

```
// App class definition
class app_c { clearAppUserData checkPermission switch };

// App type definitions
type android_t;

type app_contacts_t;

type app_launcher_t;

type app_example_t;

type app_enterprise_t;

type app_unknown_t;

defaultAppType app_unknown_t;

// App type assignments
appType android_t {
    Package:package_name=android;
    Package:package_name=com.android.keychain;
    Package:package_name=com.android.settings;
    Package:package_name=com.android.seandroid_manager;
    Package:package_name=com.android.providers.settings;
    Package:package_name=com.android.systemui;
    Package:package_name=com.android.vpndialogs;
};

appType app_contacts_t {
  Package:package_name=com.android.contacts;
};

appType app_launcher_t {
  Package:package_name=com.android.launcher;
};

appType app_example_t {
  Developer:signature=0xFEF9...;
};

appType app_enterprise_t {
    ExternalSignature:keyFileLocation=/etc/enterprise_public_key.file;
    ExternalSignature:signatureFileLocation=assets/enterprise_signature.
        file;
};
```

Listing 4: FlaskDroid policy excerpt describing Intents and corresponding operations. The class definitions `activity_c`, `intent_c` and `broadcast_c` describe operations applicable to Activities and (Broadcast) Intents. Intents are assigned the `intent_unknown_t` type unless otherwise specified [34].

```
// Activity class definition
class activity_c { start stop grantURIPermission finish moveTask };

// (Broadcast) Intent class definition
class intent_c { send receive };

class broadcast_c { send receive sendSticky receiveSticky registerReceiver
    unregisterReceiver };

// Intent type definitions
type intent_start_launcher_t;

type broadcast_intent_bootcompleted_t;

type intent_unkown_t;

defaultIntentType intent_unknown_t;

// Intent type assignment
intentType intent_start_launcher_t {
    Action:action_string=android.intent.action.MAIN;
    Categories:category=android.intent.category.HOME;
};

intentType broadcast_intent_bootcompleted_t {
    Action:action_string=android.intent.action.BOOT_COMPLETED;
};
```

ContentProviders. ContentProviders expose structured data across application boundaries via an SQL-like interface. While not mandatory, most ContentProviders are implemented via SQLite [238] databases. A ContentProvider operating as a Userspace Object Manager assigns types to the data entries it manages. It further enforces access control on create, retrieve, update and delete operations. Access control enforcement can be coarse- and fine-grained: Android's standardized ContentProvider interface makes it feasible to implement coarse-grained access control (see Listing 5). For example, FlaskDroid can restrict access to specific operations on the entire ContactsProvider to specific applications. However, our design also considers fine-grained filtering by assigning types to individual cells (e.g., name, email address, phone number or postal address of a contact), which further enables graceful enforcement by returning empty or fake data.

Services. Android Services expose functionality via remote procedure call interfaces to other applications. Service interfaces are generated from service definitions, which are specified using the Android Interface Definition Language (AIDL [134]) and define the signatures of exposed methods. Android's AIDL compiler at compilation time generates corresponding Java stub and skeleton code.

Listing 5: FlaskDroid policy excerpt describing ContentProviders and corresponding operations. The class definition `contentProvider_c` defines operations applicable to all ContentProviders. Individual ContentProviders, such as the ContactsProvider, inherit these operations. Finally, we define individual types for data structures managed by the ContactsProvider [34].

```
1  // ContentProvider class definition
2  class contentProvider_c { query insert update delete readAccess
       writeAccess };
3
4  class contactsProvider_c inherits contentProvider_c;
5
6  class calendarProvider_c inherits contentProvider_c;
7
8  // ContactsProvider type definitions
9  type contacts_name_t;
10 type contacts_email_t;
11 type contacts_phone_t;
12 type contacts_postal_t;
13 type allContactsData_t;
```

To instrument arbitrary Android Services as Userspace Object Managers we adopted an approach inspired by Dietz et al. [58]: We modified the AIDL compiler to automatically insert an access control query into generated stub code based on the caller UID, a generic `service_c` class as well as a developer-supplied method-specific type specified in the AIDL file. As demonstrated in Listing 6 FlaskDroid additionally supports manual instrumentation of Services on the code level as well by defining Service-specific operations. This approach allows a more straightforward differentiation between operations, which we show here based on Android's LocationManagerService. It further enables graceful enforcement by returning empty or fake data for individual methods.

4.2.4 *Access Control Rules*

We will now describe the syntax of our middleware-layer access control rules, which is inspired by the policy syntax adopted by SELinux/SEAndroid. In general, a rule starts with the keyword `allow` or `deny` to designate the rule type. By default, FlaskDroid denies any access control queries unless a specific `allow` rule exists. A rule defines a list of subject types it applies to, as well as relevant object types and the object class. Finally, the concrete operation(s) which should be allowed (or denied) are listed.

For example, the first rule in Listing 7 allows subject processes of type `app_launcher_t` and `android_t` to `start` Activities (class `activity_c`) of object type `app_telephony_t`, `app_launcher_t`, `app_example_t`, `app_enterprise_t` and `app_unknown_t`. The second rule (Line 3) allows all processes of type `android_t`, `app_contacts_t`, `app_launcher_t` and `app_telephony_t` to `query` the ContactsProvider (class `contactsProvider_c`) for all contacts information (type `allContactsData_t`).

Listing 6: FlaskDroid policy excerpt describing Services and corresponding operations. The class definition `service_c` defines the operations applicable to all Services. Individual Services inherit these operations. The `locationManagerService_c` class extends these operations by a set of operations specific to this particular Service [34].

```
// Service class definitions
class service_c {
    start stop bind callFunction find
};

class locationManagerService_c inherits service_c { getAllProviders
    getProviders requestLocationUpdates removeUpdates addGpsStatusListener
    sendExtraCommand addProximityAlert removeProximityAlert getProviderInfo
     reportLocation isProviderEnabled getLastKnownLocation addTestProvider
    removeTestProvider setTestProviderLocation clearTestProviderLocation
    setTestProviderEnabled clearTestProviderEnabled setTestProviderStatus
    clearTestProviderStatus
};
```

Listing 7: FlaskDroid policy excerpt showing access control rule definitions [34]

```
allow {app_launcher_t android_t} {app_telephony_t app_launcher_t
    app_example_t app_enterprise_t app_unknown_t}: activity_c {start};

allow {android_t app_contacts_t app_launcher_t app_telephony_t}
    allContactsData_t: contactsProvider_c {query};
```

4.2.5 *Context Providers*

FlaskDroid supports context-dependent access control rules which are dynamically activated or deactivated at runtime depending on the security requirements and state of the device. To this end, Context Provider plugins can be registered with the Userspace Security Server (see Figure 8). Context Providers can evaluate a variety of information available on FlaskDroid-based devices, ranging from the current geolocation, date and time to the WiFi network environment, or even currently running applications or previous policy decisions by the Userspace Security Server. When a context is activated, the Context Provider forwards this information to the Userspace Security Server, which activates or deactivates a corresponding set of rules and propagates this information to our kernelspace SecurityServer. Context Providers thus decouple the context definition and detection from corresponding access control rules.

Listing 8 demonstrates how context-aware access control is implemented using FlaskDroid in practice. In this example, a Context Provider recognizes safe environments based on the current geolocation of the device. When a device enters this location, access control rules are activated on both the middleware- and kernel layer via policy Booleans. While the middleware policy Boolean `safeEnv_b` directly activates specific middleware-

layer policies, the kernel Boolean `enable_ip_sockets_b` is propagated to the kernel layer for enforcement via SEAndroid, in this case to allow applications to use network sockets.

Listing 8: FlaskDroid policy excerpt describing context-aware access control. The context definition `safeEnv_con` defines a safe environment. When the corresponding Context Provider activates the `safeEnv_con` context, the middleware- and kernel-layer policy Booleans `safeEnv_b` and `enableIpSockets_b` are set, which enable middleware- and kernel-layer access control rules [34].

```
// Middleware-layer policy Boolean
bool safeEnv_b = false;

// Kernel-layer policy Boolean
kbool enableIpSockets_b = false;

// Context definition
context safeEnv_con;

// switchBoolean statements define how context definitions relate to policy
    Booleans
switchBoolean {
    context = safeEnv_con;
    safeEnv_b = true;
    enableIpSockets_b = true;
    auto_reverse=true;
};

// Context-dependent middleware policy rules
if (safeEnv_b) {
    ...
}
```

4.2.6 *Support for Multiple Stakeholders*

As noted before FlaskDroid allows multiple stakeholders to deploy access control policies to protect their own resources. For example, the device manufacturer could provide a basic system policy, while a user might configure additional policies to improve user privacy. Further, app developers can supply policies within their applications, which are extracted at installation time by our extensions to PackageManagerService. These policies are however limited in scope as they are only considered during access control decisions involving the developer's app components. This approach allows app developers to optionally define their own Userspace Object Managers, types and classes and register them for mediation with the Userspace Security Server via our FlaskDroid software development kit.

Policies deployed by multiple stakeholders do not necessarily align. Consequently, the Userspace Security Server has to resolve policy conflicts at runtime. Related work has presented a variety of conflict resolution strategies [197, 161], such as *all-allow* (all deployed policies have to grant access) or *any-allow* (only one deployed policy has to grant access). While FlaskDroid generally supports different reconciliation strategies, we by default assume a conservative consensus strategy where all deployed policies need to approve.

4.3 CASE STUDIES

In this section, we describe a set of practical privacy- and security protecting use-cases and how they can be instantiated using FlaskDroid.

4.3.1 *Privacy Enhanced Operating System Components*

Android applications access privacy- and security sensitive data stored on the device via designated system Services and ContentProviders. While Android provides coarse-grained protection of sensitive data via permissions it is not possible for a user to restrict access to individual data fields in system components. For example, an application can either retrieve all contacts data from the ContactsProvider or no data at all.[1] Consider for example that the popular WhatsApp instant messenger has been shown to upload the phone numbers of all contacts to its back end servers in order to identify other WhatsApp users [22]. Further, Android by default does not enforce any permissions on most sensors (e.g., accelerometer and gyroscope), and related work has shown that these sensors can be abused to derive sensitive keyboard input [290, 37].

In FlaskDroid, our modified Android SensorService operates as a Userspace Object Manager and enforces access control on sensors. Using a Context Provider which monitors the state of the on-screen keyboard and a corresponding access control policy FlaskDroid enforces that access to sensors is prohibited while the on-screen keyboard is active. Further, a user-defined policy instructs the ContactsProvider Userspace Object Manager to only expose contacts data of friends and families to WhatsApp while filtering out any enterprise contacts. A graphical user interface inspired by our myTunes architecture [30] provides a user-friendly way to configure access control rules within the policy without the hassle of developing corresponding type enforcement policies.

4.3.2 *Privacy Enhanced Image Media Store*

Modern Android smartphones are equipped with cameras. Photos a user takes may contain sensitive information, where the sensitivity is defined by the current usage context of the device. For example, in case a device is used for enterprise and private purposes, photos taken while located on company premises or during working hours should not be accessible by private apps. In addition, a user may want to protect his private photos from being accessed by the employer.

When a photo taken by the user is stored on the device, meta-information about the photo (filename, location etc.) and the photo file itself are accessible by all apps which can access the external storage area. While meta-information is stored in the Mediaprovider ContentProvider, the photo itself is stored on the external storage area of the device, which is implemented using either an embedded flash module or a removable SD card. Android uses the VFAT file system for this storage area, which does not provide fine-grained access

[1] Android's URI permissions, which are essentially capabilities on individual ContentProvider records, are a notable exception. However, they are non-generic since they require developer participation.

control. Recent Android versions emulate this VFAT file system by means of a *File System In Userspace (FUSE)* driver [85]. A reference monitor in the kernel checks Linux GID based permissions (`READ_EXTERNAL_STORAGE` / `WRITE_EXTERNAL_STORAGE`) whenever apps try to access this storage area. Thus, all apps which hold these permission can access all photos.

In FlaskDroid, the sensitivity of a photo is defined by the usage context of the device while the photo was taken. This information can either be derived from user input by asking the user whenever a photo is taken, or by context derivation from sensor information (see Section 4.2.5). We assign a corresponding type to the photo metadata stored by the camera app in the Mediaprovider, which acts as a Userspace Object Manager.

A technical challenge for access control on the photo files themselves is the fact that the VFAT file system does not support extended file system attributes, a prerequisite for storing the SELinux metadata (e.g., types) for file objects. When the VFAT file system is emulated by the previously described FUSE module, this module can be instrumented to act as a Userspace Object Manager and to mediate access based on the application and file type. Alternatively, a file system with support for extended attributes for the external storage area, as proposed by related work [66], can be used.

It should be noted that when using removable media FlaskDroid's access control mechanisms can be circumvented by removing the SD Card and accessing it from another device. This challenge can be addressed by encrypting the contents of the SD Card with a key bound to the mobile device.

4.3.3 *Phone Booth Mode*

A user may want to temporarily give his smartphone to another person for the purpose of making a phone call. In this situation the user provides physical control over his device to a person he not necessarily completely trusts. This person can access security- and privacy sensitive data while he has physical access to the phone.

The goal of this use-case is to temporarily lock the device in a secure state in which it can be handed out to another person for the sake of making a call. In this mode, the phone is configured to only grant access to telephony-related features and apps (see Figure 9). Further, access to privacy-sensitive data stored on the phone, such as contacts and call log entries, is denied.

FlaskDroid's fine-grained access control within ActivityManagerService enables us to decide whether or not specific applications can be moved into or off the foreground. FlaskDroid allows the user to activate the "phone booth mode" context by pressing a button. While this context is active, the access control policy enforces that only the dialer app is in the foreground. Further, access control rules state that no applications can access the ContactsProvider or CallLogProvider, as shown in Figure 9. Finally, to leave phone booth mode the device owner needs to authenticate himself towards the device, for example by entering his lockscreen PIN number.

Listing 9 presents a summary of the relevant policy. This policy defines a separate Boolean `phoneBooth_b` (Line 1) to represent the phone state, i.e., either the phone booth mode is activated or not. We use a dedicated context `phoneBooth_con` (Line 3) and corresponding switchBoolean statement (Lines 5-10) to manage this context. Our policy defines

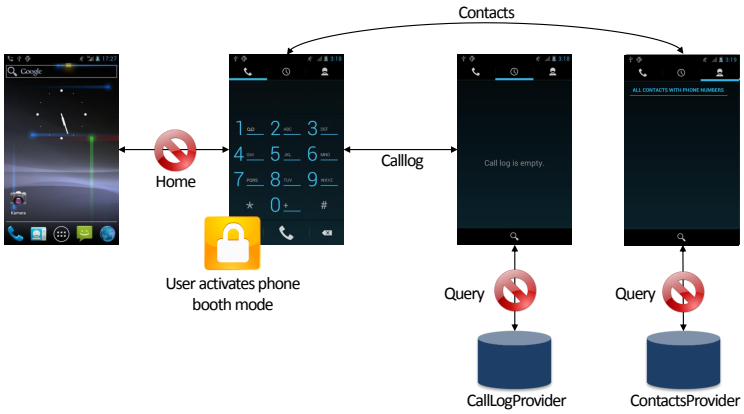


Figure 9: Phone booth mode

that on activation of this switchBoolean statement the `phoneBooth_b` Boolean is set to true, and that it automatically resets to false when the `phoneBooth_con` context is deactivated. The snippet shows the rules for normal operation of the phone (Lines 12-26), including querying contacts, switching Activities, etc. However, several operations are only allowed if the phone is *not* in phone booth mode (Lines 28-33), meaning that these rules are disabled if phone booth mode is active (i.e., the `phoneBooth_b` Boolean is true). The affected rules address switching of Activities and querying for contacts and call log data. Since the phone booth mode is activated from within the phone app, which is shown in the foreground, deactivating these rules forces the phone app to stay in foreground and prevents it from accessing contacts and call log data.

It should be noted that Android version 5 introduced a similar feature, denoted *screen pinning* [99], which locks the user into one specific application until the device owner unlocks the device. Our FlaskDroid-based instantiation predates this feature and we implemented it merely using a corresponding access control policy and Context Provider.

4.3.4 *App Developer Policies (Saint)*

Saint [180] is an Android security extension which allows app developers to distribute access control policies with their applications in order to protect their app components from illegitimate access by third-party apps. To do so, Saint mediates Binder-based inter process communication (IPC) using policies based on Source (calling app component and Intent message), Destination (callee app component), optional Conditions applying to source- and destination components (e.g., granted permissions) and the State of the device (e.g., state of network interfaces or geolocation). At runtime, these policy rules are evaluated by Saint's policy decision point and enforced on Binder IPC transactions.

To demonstrate their approach the authors define a payment scheme, where a shopping app ships with a policy which restricts interaction of this shopping app with external applications, such as password vault, payment or personal ledger applications. Their example policy uses Conditions to mandate that any potential ledger application must not hold

Listing 9: FlaskDroid policy excerpt implementing phone booth mode. Rules starting with the `self` keyword allow components to perform operations on themselves.

```
bool phoneBooth_b = false;

context phoneBooth_con;

switchBoolean
{
    context=phoneBooth_con;
    auto_reverse=true;
    phoneBooth_b=true;
};

self: app_c {checkPermission};
self: activity_c {finish moveTask};
self: broadcast_c {receive send};

allow {app_system_t app_contacts_t app_launcher_t} allContactsData_t:
    contactsProvider_c {query};

allow {app_system_t app_contacts_t app_launcher_t} allCallLogData_t:
    calllogProvider_c {query};

allow {app_system_t app_telephony_t app_contacts_t app_launcher_t} {
    app_system_t app_telephony_t app_contacts_t app_launcher_t}: package_c
    {getPackageInfo getPackageInfoWithUninstalled getPackageUID
    getPackageGIDs getPackagesForUid getNameForUid getUidForSharedUser
    findPreferredActivity queryIntentActivities getInstalledApplications
    getInstalledApplicationsWithUninstalled getInstalledPackages
    getInstalledPackagesWithUninstalled};

allow {app_system_t app_telephony_t app_contacts_t app_launcher_t} {
    app_system_t app_telephony_t app_contacts_t app_launcher_t}: app_c {
    checkPermission};

allow {app_system_t app_telephony_t app_contacts_t app_launcher_t} {
    app_telephony_t app_contacts_t}: activity_c {start};

allow {app_system_t app_telephony_t app_contacts_t app_launcher_t} {
    app_system_t app_telephony_t app_contacts_t app_launcher_t}: activity_c
     {moveTask finish};

if(~phoneBooth_b)
{
    allow {app_system_t app_telephony_t app_contacts_t app_launcher_t} {
        app_system_t app_telephony_t app_contacts_t app_launcher_t}:
        activity_c {start moveTask finish};
    allow app_telephony_t allContactsData_t: contactsProvider_c {query};
    allow app_telephony_t allCallLogData_t: calllogProvider_c {query};
};
```

the `INTERNET` permission, and that a specific vault application must be used. Further, the policy enforces constraints on the version range of the payment application.

We implemented the Saint example using a corresponding FlaskDroid third-party developer policy. This policy is distributed with the shopping application. Within the scope of the policy specific app types are assigned to the password vault, shopping and ledger applications based on application metadata. Further, the policy assigns types to Intents exchanged between these applications. Finally, access control rules describe allowed component interactions between the shopping app and the specified payment, password vault and ledger applications via Intents. We refer to our publication [34] for a more detailed discussion of this use-case and the corresponding policy.

4.4 EVALUATION

To evaluate our architecture we implemented a FlaskDroid prototype based on SEAndroid in version 4.0.4 and deployed it on a Samsung Galaxy Nexus smartphone. In this section we focus on policy complexity, effectiveness and performance aspects.

4.4.1 *Policy Size and Complexity*

We first established a basic policy, which consists of types, classes and rules required for our default Userspace Object Managers discussed in Section 4.2.3. Similar to semi-automatic methods proposed by related work [193, 114] we decided to derive this basic policy by observing application behavior in human user trials using FlaskDroid's audit mode, where access control violations are logged but no access control rules are enforced. This approach is motivated by the following considerations: First, while Android provides an automated user interface exerciser denoted *monkey*, which generates random but valid input signals, related work has shown that such mechanisms potentially achieve a rather low coverage [89]. Second, modern static analysis tools taking into account Android's event-driven execution model were not available at the time FlaskDroid was developed [305, 89].

During our trials users were provided with preconfigured FlaskDroid devices, which contained test data (e.g., fake contacts) and had access to test accounts (e.g., Email). We deployed a handcrafted *No-allow-rule* policy on these devices, which only contains 111 subject/object types, 18 classes and 63 operations necessary for our default Userspace Object Managers (e.g., ContactsProvider, LocationManager, PackageManagerService, or SensorService), but no access control rules. Users were instructed to perform every-day tasks (e.g., web browsing, adding and managing contacts, communicating and sharing data via phone calls, SMS and MMS). Further, users were encouraged to use Android's IPC mechanisms, for example by executing complex workflows, sharing data between apps and using the clipboard.

Based on the observed access control queries we derived a set of 109 access control rules necessary for correct operation of the device. Combined with the previously described type, class and attribute definitions these rules constitute our *basic policy*. Although not directly comparable, the difference in policy complexity between this basic FlaskDroid

Policy	Types	Attributes	Classes	Permissions	Rules
SEAndroid (Master branch, checkout 12/04/2012)	232	19	84	249	1359
FlaskDroid middleware MAC (basic policy from 12/04/2012)	111	9	18	63	109
SELinux reference policy (v2.20120725, no distribution option)	661	132	81	239	278
SELinux Fedora 17 (targeted, policy.27 from 12/04/2012)	3900	313	83	248	103235
SELinux CentOS 6.3 (targeted, policy.24 from 12/05/2012)	3508	277	81	235	275791
SELinux Debian 6.0.6 (default, policy.24 from 12/05/2012)	1285	190	77	229	49159

Table 4: Overview of policy complexity: Comparison of SELinux, SEAndroid and FlaskDroid policies [34]

policy and the default SELinux policy is in the order of several magnitudes (see Table 4). This indicates that component-based architectures of operating systems for smart mobile devices facilitate the design of type enforcement policies. Both previous work [298, 166] and more recent studies analyzing both SELinux and FlaskDroid on Android version 5 and above [233, 31] confirm this observation.

4.4.2 *Effectiveness*

We evaluated the effectiveness of FlaskDroid based on the security models introduced in Section 4.3.

Root Exploits. To show that FlaskDroid can be effective against common operating-system privilege escalation attacks we verified that SEAndroid successfully mitigates the effect of the *mempodroid* root exploit. While the exploit succeeds in elevating its process to root privileges, the process is constrained by the underlying SE Android policy to the limited privileges granted to the root user [232].

Malicious Apps Executed with root Privileges. SEAndroid has been shown to constrain file-system privileges of application processes with root UID. However, on the middleware layer the root user still inherits all available permissions. To constrain the root user on the middleware level we introduced a corresponding `aid_root_t` type and derived necessary allow rules during user trials. Unsurprisingly, only one allow rule was required for stable operation, since Android by design aims to minimize the number of processes executed with root privileges.

Privacy-intrusive Applications. To verify that FlaskDroid effectively mitigates the threat of over-privileged applications we first deployed a synthetic test application which accesses the LocationManager, SensorManager and ContactsProvider. We further installed samples of the Android.Loozfon [247] and Android.Enesoluty [246] malware, which use their extensive permissions to steal sensitive user information. To test the effects of our

Attack	**Test**
Root Exploit	mempodroid Exploit
App executed by root	Synthetic Test App
Over-privileged and Information-Stealing Apps	Known malware Synthetic Test App WhatsApp v2.8.4313 Facebook v1.9.1
Confused Deputy	Synthetic Test App
Collusion Attack	Synthetic Test Apps [218]

Table 5: List of attacks considered in our testbed

architecture on real-life applications we further deployed both the popular WhatsApp instant messenger as well as the Facebook client application on a FlaskDroid-enabled device. We then manually verified that a corresponding access control policy successfully prohibited any attempts by the deployed applications to access sensitive data. Further, our analysis has shown that our graceful enforcement did not cause any unexpected application crashes.

Application-Layer Privilege Escalation Attacks. As discussed in Section 3.3.1 related work has shown that the default Android operating system is susceptible to a variety of confused deputy attacks. FlaskDroid can effectively mitigate such attacks using fine-grained access control on inter-process communication. We selected a confused deputy present in previous versions of Android's Settings components [81], which allowed unprivileged applications to control Bluetooth, WiFi and GPS settings via a Broadcast Intent. We then verified that FlaskDroid prevents this Intent from being delivered to the Settings component using a tailored access control policy.

The mitigation of collusion attacks is however generally more involved. We verified that FlaskDroid can effectively mitigate specific collusion attacks by preventing inter-process communication between well-known colluding applications. However, the identification of such collusion attacks is a challenging problem itself, especially when they use covert communication channels. We discuss this aspect in more detail in Section 6.2, where we analyze to what extend system-centric access control architectures can be used to detect and scrutinize such attacks. Further, while related work has demonstrated that it is to some extend feasible to defend against such attacks via Chinese wall policies [32], achieving a low false positive rate requires fine-grained information flow analysis on the sub-process level [158].

4.4.3 *Performance*

To evaluate the performance impact of FlaskDroid we deployed both the *no-allow-rule* policy and the *basic policy* on a Samsung Galaxy Nexus device. Our results are presented in Table 6 and show mean execution time and corresponding standard deviation of policy checks at the middleware layer. Further, we present the average memory consumption of the SystemServer process containing our Userspace Security Server. When comparing our

	Mean Execution Time (in μs)	Standard Deviation σ (in μs)	Memory (in MB)
FlaskDroid			
No-allow-rule	329.50	780.56	15.67
Basic policy	452.92	4887.24	16.18
Vanilla Android 4.0.4			
Permission check	330.80	8291.80	15.98

Table 6: Performance and memory usage overhead [34]

results with a vanilla Android 4.0.4 distribution our overhead is within acceptable limits. The high standard deviation is caused by varying system load. Overall, the confidence interval for our basic policy is 99.33% for a maximum overhead of 2 ms.

Finally, we note that the performance impact of SEAndroid kernel-layer enforcement has been evaluated by related work [232]. FlaskDroid introduces no changes to the SEAndroid implementation and only minimal changes to the default SEAndroid policy depending on the use case. Therefore, these previous results apply to FlaskDroid as well.

4.5 CONCLUSION

In this chapter, we described our FlaskDroid architecture, which prototypes a policy-driven approach to mandatory access control for the Android operating system. FlaskDroid extends SELinux/SEAndroid [153, 232] type enforcement to Android's application and middleware layer and enables the policy driven instantiation of use-case specific access control solutions. Our implementation and evaluation of the FlaskDroid architecture demonstrate that Android's API-oriented design is an ideal candidate for the integration of system-centric mandatory access control.

While our evaluation shows that it is indeed possible to implement a variety of use-cases using policy-driven mandatory access control FlaskDroid is not without limitations: First, FlaskDroid imposes type enforcement on developers, which carries a reputation for being overly complex, especially when considering rather simple use-cases for access control. To some extend this argument is refuted by our evaluation and related work [298, 166, 233], which show that the resulting policies are significantly less complex then policies targeting desktop and server operating systems. Second, while the FlaskDroid architecture generally supports third-party access control policies deployed by application developers, our implementation currently only enforces these policies on the application and middleware layer. We argue that truly extensible system-centric access control architectures should also provide a strictly controlled interface which allows third parties to configure kernel-layer access control enforcement at runtime. The Android Security Modules architecture presented in the following Chapter 5 addresses these challenges by providing a *programmable* interface for extending Android's access control architecture.

5

A MODULAR AND PROGRAMMABLE ACCESS CONTROL ARCHITECTURE

The previously introduced FlaskDroid architecture implements a policy-driven approach to bring extensible access control to the Android platform. However, history has shown that only providing one specific access control solution, for example type enforcement or capabilities, does not meet the demands of all potential OS customers (e.g., consumers, enterprise or government). Therefore, a truly extensible OS security interface should be *programmable* [278].

The goal of the Android Security Modules (ASM) framework presented in this chapter is to promote such *programmable* OS security extensibility on the Android platform. By providing a programmable interface, ASM enables an extensible access control that allows not only type enforcement, but also novel security models not yet invented. In short, ASM seeks to accomplish for Android what the Linux Security Modules (LSM) [283] and TrustedBSD [277] frameworks have provided for Linux and BSD, respectively: Our ASM framework provides a set of authorization hooks to build reference monitors for Android security. We design and implement an open source version of ASM within Android version 4.4.1 and empirically demonstrate negligible overhead when no security module is loaded. ASM fulfills a strong need in the research community. It provides researchers a standardized interface for security architectures and will potentially lead to field enhancement of devices without modifying the system firmware (e.g., BYOD), if adopted by Google.

Contribution. To summarize, our main contributions are as follows:

- **Programmable Interface for System-centric Access Control Enforcement.** Our ASM framework brings *programmable* OS security extensibility to Android. It allows multiple simultaneous security modules, denoted ASM Apps, to enforce security requirements on the kernel, middleware and application layer.
- **Evaluation of Performance and Energy Consumption.** By dynamically activating access control hooks based on the security requirements of individual ASM Apps our framework minimizes performance overhead.
- **Development of Use-Case Specific Security Solutions.** We demonstrate the flexibility of our ASM framework by instantiating a set of security extensions proposed by related work as ASM Apps.

The remainder of this chapter proceeds as follows. Section 5.1 introduces domain-specific background knowledge. We describe the design and implementation of our ASM framework in Section 5.2, followed by a discussion of practical use-cases we implemented using ASM in Section 5.3. Section 5.4 evaluates the performance and energy consumption overhead of the ASM architecture. Finally, Section 5.5 concludes.

Remark. The results presented in this section were achieved in collaboration with Adwait Nadkarni, William Enck and Ahmad-Reza Sadeghi. The initial idea was developed during discussions between all involved authors. Adwait Nadkarni and the author contributed equally to the design and implementation of the ASM architecture. The author was responsible for evaluation of performance and energy consumption. Use-cases "MockDroid" and "AppLock" were implemented by Adwait Nadkarni, while the author contributed the "App-specific Firewalling" use-case as well as additional use-cases described in individual publications (see Sections 6.1 and 6.2). William Enck and Ahmad-Reza Sadeghi were involved in fruitful discussions and general writing tasks, which improved the quality of our publication [119].

5.1 BACKGROUND

Our ASM framework follows the methodology of the Linux Security Modules (LSM) [283] and TrustedBSD [277] reference monitor interface frameworks. Both frameworks have been highly successful and allow the programmatical instantiation of custom reference monitors. In Linux, LSM is widely used to extend Linux security enforcement. Version 4.5 of the Linux kernel source includes SELinux [153], AppArmor [21], Tomoyo [114], SMACK [216], and Yama [1] LSMs. Unsurprisingly, both LSM and TrustedBSD have been embraced by platform providers for mobile operating systems. As noted in Section 2.3, Android uses SELinux since Version 4.3 [101], while the Tizen OS relies on SMACK. TrustedBSD is not only used by FreeBSD, but also by Apple to implement kernel-level sandboxing in iOS and Mac OS X [278].

We will now briefly discuss the LSM architecture. In short, LSM provides a uniform hook-based interface for the implementation of custom access control models within the Linux kernel. These hooks are deployed within security-sensitive kernel subsystems, such as the virtual file system, process and memory management. In LSM, access control models are developed in C code and encapsulated as modules within the Linux kernel. They are integrated into the operating system at compile time. In mainline Linux as well as Android's modified Linux kernel only one of these modules can be active. Upon platform initialization, this active module registers for hooks it is interested in mediating at runtime. Figure 10 depicts the mediation process. When an application process performs a security-sensitive operation, such as opening a file (Step 1), the Linux kernel first evaluates corresponding discretionary access control (DAC) rules (Step 2). In case the DAC rules allow the operation to proceed, a LSM hook in the relevant subsystem invokes the LSM framework (Step 3), which in turn triggers a callback to the active security module (Step 4). This module then decides whether or not the operation is allowed to proceed and can thus override positive decisions by the DAC framework, making it an ideal platform to implement mandatory access control models.

However, a fundamental limitation of the LSM architecture is that it only mediates sensitive operations on the kernel layer. Operating systems for smart mobile devices, such as Android, however implement a rich userland API, which is located on the middleware and application layer and offers access to security- and privacy-sensitive high-level services and resources to applications. The previously discussed FlaskDroid architecture (see Chap-

ter 4) has shown that kernel-level mediation alone is insufficient due to the semantic gap between kernel-, middleware- and application-layer structures. To address this limitation, our ASM framework extends the idea of modular and programmable access control from the Linux kernel to Android's middleware- and application-layer.

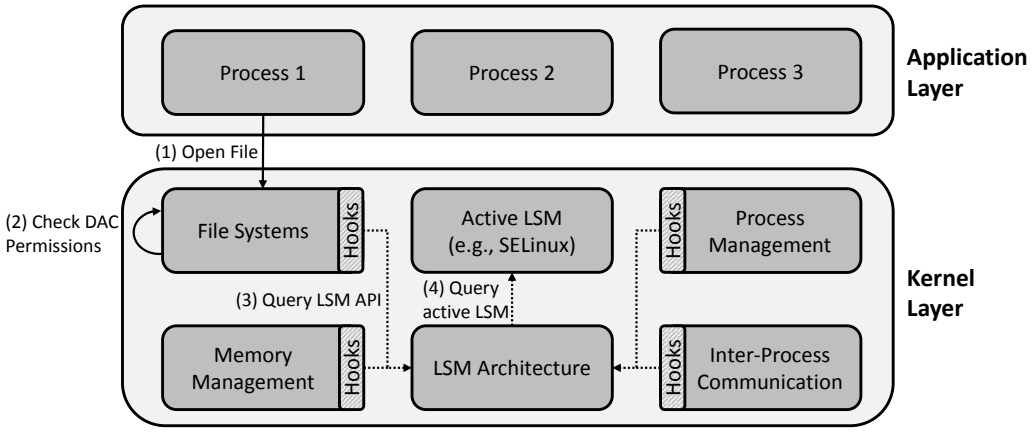


Figure 10: Linux Security Modules (LSM) architecture

5.2 ASM ARCHITECTURE

ASM provides a unified interface for building new reference monitors for the kernel-, middleware- and application-layer. By doing so, ASM allows reference monitor developers to focus on their novel security enhancements and not on placing hooks correctly. It also allows separate scrutiny of authorization hook placement that benefits all reference monitors built on top of ASM.

Figure 11 shows the ASM framework architecture. Reference monitors are implemented as ASM Apps. Each ASM App registers for a unique set of authorization hooks, specifying a callback for each. The ASM Bridge manages registered ASM Apps and receives access control decision queries (also denoted *protection events*) from authorization hooks placed throughout the Android OS. Whenever a hook is invoked, the ASM Bridge queries all active ASM App callbacks. Since Android places functionality in multiple userspace processes, authorization hooks only query the ASM Bridge if the hook is explicitly enabled. ASM also supports authorization hooks within the Linux kernel as well as third-party applications. To achieve kernel authorization, a special Linux Security Module, denoted ASM LSM, performs upcalls to the ASM Bridge, again only doing so for explicitly enabled hooks.

5.2.1 *ASM Apps*

Reference monitors are built as ASM Apps. They are developed using the same conventions as other Android applications. The core part of an ASM App is a Service component that implements the reference monitor hook interface provided by ASM. There are three main

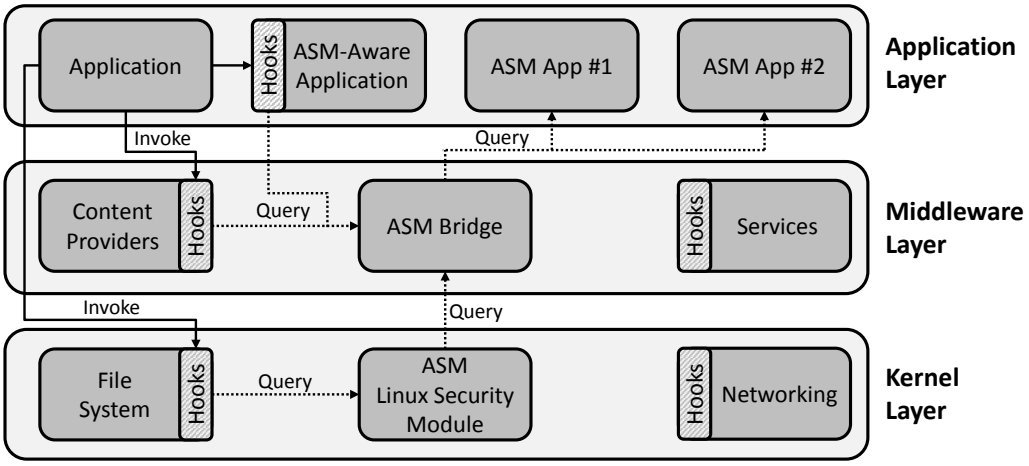


Figure 11: ASM framework architecture

functionalities that must be provided within this Service. Finally, the registration interface of the ASM Bridge is protected by Android permissions.

ASM App Registration. An ASM App must register itself with the ASM Bridge after it is installed. The time of registration depends on logic in the specific ASM App. For example, the ASM App could register itself automatically after installation, or it could provide a user interface to enable and disable it. When the ASM Bridge receives the registration, it updates its persistent configuration. To activate the ASM App, the device must reboot. We require a reboot to ensure ASM Apps receive all protection events since boot, which may impact their protection state.

Hook Registration. The ASM App Service component is started by ASM during the boot process. At this time, the ASM App registers for reference monitor interface hooks for which it wishes to receive callbacks. Different hooks incur different overheads. ASM only enables a reference monitor hook if it is registered by an ASM App. Therefore, ASM App developers should only register for the hooks required for complete mediation. Finally, if the ASM App registers for hooks defined by a third-party application (see Section 5.2.4), the application developer and the ASM App developer must agree on naming conventions.

Handling Hook Callbacks. Once an ASM App registers for a reference monitor interface hook, it will receive a callback whenever the corresponding protection event occurs. The information provided in the callback is hook-specific. The ASM App returns the access control decision to the ASM Bridge. Some hooks allow the callback to replace data values, which is required for graceful enforcement (see Section 3.5.2). Finally, similar to registration for third-party hooks, the ASM App developer must coordinate with the application developer for information passed to the callback.

Registration Protection. Reference monitors are highly privileged. While ASM does not allow an ASM App to override existing Android security protections, ASM must still protect the ability to receive callbacks. ASM protects callbacks using Android's existing permission model. It defines two permissions: `REGISTER_ASM` and `REGISTER_ASM_MODIFY`.

The ASM Bridge ensures that an ASM App holds the `REGISTER_ASM` permission during both ASM App registration and hook registration. Finally, since replacing data values in an access control callback has greater security implications, the ASM Bridge ensures the ASM App holds the `REGISTER_ASM_MODIFY` permission if it registers for a hook that allows data modification. This allows easy ASM App inspection to identify its abilities.

ASM App Deployment. How the ASM permissions are granted has a significant impact on the practical security of devices. Previous studies [80] have demonstrated that end users frequently do not read or understand Android's install time permissions. Therefore, malware may attempt to exploit user comprehension of permissions and gain ASM App privileges. In general, our architecture can support a variety of ASM App deployment models. In the use case where researchers change AOSP source code, these permissions can be bound to the firmware signing key, thereby only allowing the researchers' ASM Apps to be granted access. In the case where ASM is deployed on production devices, ASM could follow the security model used by the mobile device management API. That is, a secure setting that is only modifiable by users would enable whether ASM Apps can be used. An alternative is to use a model similar to Android's "Unknown sources" setting for installing applications from alternative sources. That is, unless a secure user setting is selected, only Google certified ASM Apps can be installed.

5.2.2 *ASM Bridge*

The ASM Bridge 1) provides the reference monitor interface, and 2) coordinates protection events that occur in authorization hooks placed throughout the Android OS, as well as third-party applications. As discussed in Section 5.2.1, ASM Apps notify the ASM Bridge of their existence via an ASM App registration followed by individual hook registrations. We now discuss important reference monitor interface considerations.

Per-Hook Activation. All reference monitor interface hooks are deactivated by default. Each authorization hook maintains an activation state variable that determines whether or not the ASM Bridge is notified of protection events. This approach eliminates unnecessary inter-process communication (IPC) and therefore improves performance when no ASM App requires a specific hook. Likewise, this approach allows ASM to achieve negligible overhead when no ASM Apps are loaded (see Section 5.4.1).

When an ASM App registers a callback for a deactivated hook, the ASM Bridge activates the hook by notifying the corresponding authorization hook implementation. ASM maintains a list of active hooks in each OS component (e.g., OS Service component, OS ContentProvider component). When a hook is triggered, the OS component creates a corresponding protection event that is sent to the ASM Bridge. When the ASM Bridge receives the protection event for a hook, it is forwarded to each ASM App that registered for the hook. Similarly, the ASM LSM in the kernel maintains a separate activation state variable per hook and performs an upcall for each protection event.

Callback Timeouts. The ASM Bridge is notified of protection events via synchronous communication. Authorization hooks in userspace communicate with the ASM Bridge using Binder IPC, and the ASM LSM uses synchronous upcalls, as described in Section 5.2.4.

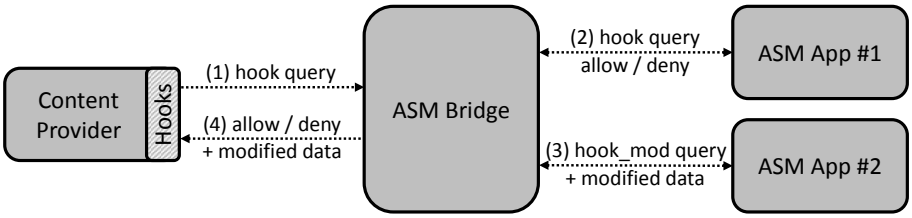


Figure 12: ASM hook invocation

The ASM Bridge then uses synchronous Binder IPC to invoke all ASM App callbacks for the hook corresponding to the protection event. If the ASM App callback implementation is buggy, the authorization hook may stall execution. Therefore, ASM has the ability to set timeouts on callback execution. If a timeout occurs, the ASM Bridge conservatively assumes access is denied.

Master Policy. ASM supports multiple simultaneous ASM Apps, which is motivated by multi-stakeholder scenarios, e.g. users, administrators, and device manufacturers installing ASM Apps on a device. When more than one ASM App is active, a reconciliation strategy is required to handle potential conflicts between access control decisions. The correct conflict resolution strategy is highly use-case specific. Therefore, providing a general solution is infeasible [34].

ASM addresses this problem using a master policy that defines policy conflict reconciliation. For our implementation and evaluation, we use a consensus strategy. That is, all active ASM Apps must grant an access control decision for an action to be allowed. Similar to FlaskDroid [34], the master policy can be easily modified to support other conflict resolution strategies [197, 161]. For example, a priority-based resolution policy hierarchically orders ASM Apps, and a voting policy allows an action if a specified threshold of ASM Apps grant it.

5.2.3 *Callbacks Modifying Data*

Before discussing the reference monitor interface hooks provided by ASM, we must describe one last concept. While most ASM Apps require a simple allow/deny access control interface, some may benefit from the ability to modify data values. For example, MockDroid [24] modifies values (e.g., IMEI, location) returned by OS APIs before they are sent to applications to enable *graceful* access control enforcement without causing application crashes. ASM supports data modifications by providing a special hook type.

Each reference monitor interface hook that potentially requires data replacement is split into two variants: 1) *normal*, which allows the corresponding callback to simply allow or deny the event, and 2) *modify*, which allows the corresponding callback to modify the value returned by the API, in addition to specifying allow or deny. As mentioned in Section 5.2.1, modifying data has a greater security sensitivity, and therefore registration of a *modify* callback requires the `REGISTER_ASM_MODIFY` permission.

Figure 12 shows how the ASM Bridge manages normal and modify hooks. To reduce the overhead of handling authorization hooks, the ASM Bridge is only notified once per

Listing 10: Example callback prototypes modifying data

```
1 // Callback received by the ASM Bridge
2 int start_activity(inout Intent intent, in String resolvedType, in
      ActivityInfo act, int requestCode, int callingPid, int callingUid);
3
4 // Callback to individual ASMs (No modify data)
5 int start_activity(in Intent intent, in String resolvedType, in
      ActivityInfo act, int requestCode, int callingPid, int callingUid);
6
7 // Callback to individual ASMs (Modify data)
8 int start_activity_mod(in Intent intent, inout Bundle extras, in String
      resolvedType, in ActivityInfo act, int requestCode, int callingPid, int
       callingUid);
```

protection event. The ASM Bridge then manages the normal and modify versions, returning the access control decision and modified data value (if needed) to the authorization hook. Additionally, the ASM Bridge invokes all of the normal callbacks before the modify versions. This approach allows a performance improvement if a consensus master policy is used (see Section 5.2.2). In this case, if a normal hook denies access, the modify callbacks do not need to be called.

Example 1. Listing 10 explains this distinction further via example. The listing shows the callback prototypes for the `start_activity` protection event. The first prototype shown, `start_activity`, is the ASM Bridge callback used by the authorization hook in the Activity Stack subsystem of Android's ActivityManagerService. This hook is invoked after Intent resolution but before the chosen Activity component is started. The hook includes 1) the Intent message from the caller, 2) information about the Activity to be started, 3) the caller's identity, and 4) additional information for the current event. By marking the *intent* parameter as *inout* (a directive defined in the *Android Interface Definition Language* [134]), the ASM Bridge can modify it.

The ASM Bridge splits `start_activity` into the normal and modify versions. To ensure restrictive enforcement, ASM Apps can modify only the *extras* field supplied by the caller. It cannot modify information that has been reviewed by the user or the OS, such as the action string or the target Activity. To ensure this restriction, the ASM Bridge makes the Intent immutable, but supplies a mutable *Bundle* of extras extracted from the Intent to the ASMs registered for the modify data hook. The modified extras received by the ASM Bridge are then set back to the Intent before the initial callback from the Activity Stack to the ASM Bridge returns.

5.2.4 *Hook Types*

ASM provides a reference monitor interface for authorization hooks placed throughout the Android OS. We now describe five general categories of hooks: 1) lifecycle hooks, 2) OS Service hooks, 3) OS ContentProvider hooks, 4) third-party app hooks, and 5) LSM hooks.

Listing 11: Resolve Activity hook

```
// Callback received by the ASM Bridge
int resolve_activity_mod(inout List<ResolveInfo> resolvedList, in String
    resolvedtype, int userId, inout Intent intent, int callingPid, int
    callingUid);

// Callback to individual ASM apps (Modify data)
int resolve_activity_mod(inout List<ResolveInfo> resolvedList, in String
    resolvedtype, int userId, in Intent intent, int callingPid, int
    callingUid, inout Bundle extras);
```

Lifecycle Hooks. ASM provides reference monitor hooks for component lifecycle events in the ActivityManagerService (AMS), the AMS subsystems, and the PackageManagerService (PMS). Hooks in this category include: resolving Intents, starting activities and Services, binding to Services, dynamic registration of Broadcast Receivers, and receiving Broadcast Intents. We demonstrate the lifecycle hook category with the following example. Note that Example 1 is also a lifecycle hook.

Example 2. The `resolve_activity` protection event occurs within PackageManagerService. The ASM authorization hook for `resolve_activity` is placed in the PMS after the Intent has been resolved by the OS, but before a chooser with the resolved Activities is presented to the user. This hook is motivated by systems such as Saint [180] and Aquifer [168], which refine the list of resolved applications based on access control policies. Note that refining the chooser list requires data modification, and therefore, `resolve_activity` is one of few hooks that only provide a modify version.

Listing 11 shows the callback prototypes defined for `resolve_activity`. The callback received by the ASM Bridge from the Android OS contains the list of resolved components. The ASM Bridge then executes an RPC to the ASM App callbacks registered for this hook. The RPC provides a modifiable resolved component list and Bundle extras. The other parameters are immutable. It is important to prevent the ASM from adding new apps to the list, thereby overriding the OS's restrictions. Therefore, we compute the set intersection of the original list and the modified list, and return the result to the authorization hook. When multiple ASM Apps register for this hook, the ASM Bridge calls the hook callback for each ASM App, providing the modified data from the previous invocation as input.

OS Service Hooks. Lifecycle hooks include mediation for inter-component communication using Intent messages. However, ASM Apps also require mediation for OS APIs providing functionality such as getting the geographic location and taking pictures. Android implements this functionality in different Service components designated as *system Services*, e.g., LocationManagerService and SensorService.

ASM uses Android's AppOps subsystem [147] to place the authorization hooks for many OS Service hooks. AppOps was introduced to AOSP in version 4.3 and adds authorization hooks throughout the Android OS. While there have been several popular media stories of hobbyist developers using AppOps to control per-application permissions, AppOps remains

Listing 12: AppOps hook for sending SMS

```
// Callback received by the ASM Bridge
int appops_query(int opcode, int callingUid, String packageName);
// Here, opcode = OP_SEND_SMS

// Callback to individual ASMs
int send_sms(int callingUid, String packageName);
```

largely undocumented and is not yet available for public use. Based on our code inspection, AppOps appears to be an effort by Google to provide more flexible control of permission related events. Surprisingly, the permission revocation feature introduced in Android 6.0 is however not based on AppOps. Conceptually, AppOps is an Android security enhancement and could be implemented as an ASM App.

The ASM authorization hooks for Services use the AppOps syntax. AppOps defines opcodes for different operations, e.g., `OP_READ_CONTACTS` or `OP_SEND_SMS`. To identify the application performing an operation, the Linux UID and the package name of the application are used. ASM uses a single authorization hook in AppOps to call the ASM Bridge. The ASM Bridge decodes the opcode and translates it into an ASM hook.

AppOps supports graceful enforcement. That is, it returns empty data instead of throwing a Security Exception wherever possible. As a result, apps do not crash when they are denied access to resources. On the other hand, AppOps does not allow data values to be modified at runtime. Therefore, ASM adds specific data modification hooks. We also needed to extend AppOps with several hooks for privacy sensitive operations (e.g., `getDeviceId()`, `onLocationChanged()`). We now discuss two examples, including both regular AppOps hooks and ASM's data modification hooks.

Example 3. Listing 12 shows the callback prototype for the AppOps hook for sending an SMS (`OP_SEND_SMS`). The ASM Bridge receives the generic `appops_query` callback and translates the opcode to the `send_sms` hook. ASM Apps registered for the `send_sms` hook receive a callback whenever an SMS message is sent.

Example 4. Listing 13 shows the callback prototype for the `getDeviceId()` OS API call in the PhoneSubInfo (i.e., telephony) Service. The ASM Bridge receives a callback from the authorization hook and executes the `get_device_id_mod` callback in ASM Apps. ASM Apps receiving this callback can return *deny* or *allow*. If the return value is *allow*, the ASM App can also place a custom value in the first index of the `device_ids` array. This value will be sent to the Android application that invoked `getDeviceId()`, instead of the real device ID.

Content Provider Hooks. ContentProvider components are daemons that provide a relational database interface for sharing information with other applications. The ASM Bridge receives callbacks from the OS ContentProvider components (e.g., CalendarProvider, ContactsProvider, and TelephonyProvider). Separate hooks are required for the insert, update, delete and query functions. Authorization hooks for insert, update and delete must

Listing 13: `getDeviceId()` hook

```
// Callback received by the ASM Bridge
int get_device_id(int callingUid, out String[] device_ids);

// Callback to individual ASMs (Modify data)
int get_device_id_mod(int callingUid, out String[] device_ids);
```

Listing 14: CallLogProvider query hook

```
// Callback received by the ASM Bridge
int calllog_query(inout ASMCursor cursor, in Uri uri, in String[]
    projection, in String selection, in String[] selectionArgs, in String
    sortOrder, int callingUid, int callingPid);

// Callback to individual ASMs (No modify data)
int calllog_query(in ASMCursor cursor, in Uri uri, in String[] projection,
    in String selection, in String[] selectionArgs, in String sortOrder,
    int callingUid, int callingPid);

// Callback to individual ASMs (Modify data)
int calllog_query_mod(inout ASMCursor cTemp, in Uri uri, in String[]
    projection, in String selection, in String[] selectionArgs, in String
    sortOrder, int callingUid, int callingPid);
```

be invoked *before* the action is performed, to preserve the integrity of the ContentProvider's data. In contrast, the query function's hook is invoked *after* the execution, to allow filtering of the returned data.

The ContentProvider query RPC returns a database Cursor object. The Cursor object is not a *parcelable* type, and therefore the entire query response is not returned to the caller in a single Binder message. Therefore, ASM Apps cannot filter the query. To account for this, we extract the Cursor contents into a parcelable ASMCursor wrapper around a CursorWindow object to include in the callback to the ASM Bridge.

The following example demonstrates the query interface. ASM currently only provides normal (i.e., no data modification) hooks for insert, delete, and update.

Example 5. Listing 14 shows the callback prototypes for the CallLogProvider OS ContentProvider. The ASM Bridge receives the original query and the result wrapped in an ASMCursor. The callback is split into normal and modify hook variants. ASM Apps that register for the normal hook get read access to the query and the result. ASM Apps registered for the data modify hook can also modify the ASMCursor object. Both the hooks return allow and deny decisions via the return value.

Finally, we note that this use of a CursorWindow object to copy the entire ContentProvider query response into the ASM hook may lead to additional overhead when query responses are large. This is because Android uses a lazy retrieval of Cursor contents, only transferring portions of the response over Binder IPC as needed. One way to improve ASM

Listing 15: Third party hooks

```
// Callbacks received by the ASM Bridge
int hook_handler(in String name, in Bundle b);
int hook_handler_mod(in String name, inout Bundle b);

// Callback to individual ASMs (No modify data)
int hook_handler(in String name, in Bundle b);

// Callback to individual ASMs (Modify data)
int hook_handler_mod(in String name, inout Bundle b);
```

query performance is to intercept the actual data access via Binder to modify data, rather than serializing the entire response. However, this will increase the number of callbacks to ASM Apps, resulting in a trade-off.

Third Party Hooks. ASM allows third-party Android applications to dynamically add hooks to the ASM Bridge. These hooks are valuable for extending enforcement into Google and device manufacturer applications (which are not in AOSP), as well as third-party applications downloaded from application markets (e.g., Google Play Store). Third-party hooks are identified by 1) a hook name, and 2) the package name of the application implementing the authorization hook. The complete hook name is a character string of the format `package_name:hook_name`. This naming convention provides third parties with their own namespaces for hooks. Note that third parties do not specify their package name; ASM obtains it using the registering application's UID received from Binder.

To receive callbacks for third-party hooks, ASM Apps implement two generic third-party hook methods, shown in Listing 15. One method handles normal hook callbacks; the other method handles data modification hook callbacks. When the third-party application's authorization hook calls the ASM Bridge callback, it passes a generic Bundle object. The ASM Bridge forwards the Bundle to registered ASM Apps for access control decisions. As with other ASM authorization hooks, third-party hooks are only activated when an ASM App registers for it.

ASM Apps receive hook callbacks for all of their registered third-party hooks via a single interface (technically two callbacks, as shown in Listing 15). Within this callback, ASM Apps must identify the third-party hook by name and must interpret the data in the Bundle based on the third-party application's specification. We assume that ASM Apps that register for third-party hooks are aware of the absolute hook name and the contained attributes. The ASM App returns allow, deny, or allow along with a modification of the Bundle (for data modification hooks).

Finally, the third-party application developer must implement a special Service component to receive hook activation and deactivation callbacks from the ASM Bridge. The ASM Bridge sends messages to this Service to update the status of a hook. Third-party application developers must follow the message codes exposed by the ASM framework for proper hook management.

LSM Hooks. ASM Apps sometimes require mediation of kernel-level objects such as files and network sockets. ASM cannot define authorization hooks for such objects in the userspace portion of the Android OS. Instead, authorization hooks must be placed in the Linux kernel. Fortunately, the Linux kernel already has the LSM reference monitor interface for defining kernel reference monitors. For example, `file_permission` and `socket_connect` LSM hooks mediate file and network socket operations, respectively.

The main consideration for ASM is how to allow ASM Apps to interface with these LSM hooks. Several potential approaches exist. First, ASM could allow ASM Apps to load LSM kernel modules directly. This approach is appropriate when the ASM App developer also has the ability to rebuild the device firmware. For example, one target audience for ASM is security researchers prototyping new reference monitors. In this case, the ASM App developer can create userspace and kernel components and provide communication between the two.

However, we would like to also allow ASM Apps to mediate kernel-level objects without rebuilding the device firmware. Therefore, a second option is to develop a small mediation programming language that is interpreted by an ASM LSM. In this model, the ASM App developer programs access control logic within the interpreted language, and the logic is loaded along with the ASM on boot. Using an interpreted language would ensure kernel integrity.

Our current implementation uses a third option. We define a special ASM LSM that implements LSM hooks and performs synchronous upcalls to the ASM Bridge to complete the access control decision. Consistent with the rest of the ASM design, the upcall is only activated when an ASM App registers for the corresponding reference monitor hook. To integrate our ASM LSM into the kernel without removing SEAndroid, we used an unofficial multi-LSM patch [217]. We implemented authorization hooks for many commonly used LSM hooks, including `file_permission` and `socket_connect`.

While the upcall approach initially sounds like it would have very slow performance, our key observation is that many ASM Apps will require very few, if any, LSM hooks. For example, an ASM App for Aquifer [168] would only require the `file_permission` and `socket_connect` LSM hooks. Section 5.4.1 shows that both of the aforementioned hooks can be evaluated in userspace with reasonable performance overhead. Furthermore, placing all ASM App logic in one place (i.e., userspace) simplifies reference monitor design.

To improve access performance for large files, we implemented a cache with an expiration policy, where file accesses (euid, epid, inode, access_mask) and decisions received from ASM Apps on those accesses are cached; and are invalidated if the accesses do not repeat within a timeout period of 1 ms. Since we cache and match the file inode as well as the accessing subject's effective Linux user and process id (euid and epid), we do not provide an attacker the opportunity of taking advantage of a race condition (i.e., requesting for a file less than 1 ms after its access is granted).

Note that this approach may lead to a case where file access control is too coarse grained for a particular ASM App. For example, consider a situation where an application on the device reads a file continuously. An ASM App grants this application access, but if at some point during these accesses it wants to deny the access to this file, the `file_permission` hook is not triggered since the file is read before the timeout expires resulting in cache hits.

To address this problem, we allow ASM Apps to set this timeout. If multiple ASM Apps set a timeout, the master policy can determine the timeout, e.g., the smallest timeout. ASM Apps may also disable the cache, which provides all file access control callbacks to the ASM, but also degrades the performance of file reads.

5.2.5 *ASM LSM*

Finally, the ASM LSM provides two security features in addition to the LSM hook upcalls. First, it implements the `task_kill` LSM hook to prevent registered ASM Apps from being killed. Second, it implements the `inode_*xattr` LSM hooks to provide ASM Apps access to their own unique extended attribute (xattr) namespaces. That is, an ASM App can use file xattrs with a prefix matching its package name. No other applications can access these xattrs. File xattrs are needed by security enhancements such as Aquifer [168].

5.3 CASE STUDIES

In this section, we evaluate the utility of ASM by implementing existing security solutions as ASM Apps. We implement and study three examples: 1) MockDroid [24], 2) AppLock [63] and 3) DroidWall [308].

5.3.1 *MockDroid*

MockDroid [24] is a system-centric security extension for the Android OS that allows users to gracefully revoke the privileges requested by an application without the app crashing. To do so, MockDroid provides a graphical user interface that allows the user to decide whether individual applications are presented real or fake responses when accessing sensitive system components.

Original Implementation. MockDroid extends Android's permissions model for accessing sensitive operating system components by providing alternative "mock" versions. When users install an application, they choose to use the real or mock version of permissions. Users can also revise this decision later using a graphical user interface. MockDroid stores the mapping between applications and permissions in an extension to Android's PackageManagerService. This policy store is the primary policy decision point in MockDroid.

MockDroid places enforcement logic in relevant Android OS components, as well as the kernel. If an application is assigned a mock permission, the Android OS component will return fake information. For example, if an application attempts to get the device IMEI, and it is assigned the mock version of `READ_PHONE_STATE`, then the telephony subsystem will return a fake static IMEI instead of the device's real IMEI.

MockDroid also modifies the Linux kernel with enforcement logic. Recall from Section 2.3.2 that some permissions are enforced in the kernel based on Linux Group IDs (GIDs) assigned to applications. MockDroid defines additional GIDs for mock permissions enforced via GIDs. For example, if the user assigns the mock version of the `INTERNET` permission to an application, it is assigned to the `mock_inet` group instead of the `inet`

Table 7: Hooks registered by the MockDroidASM app

Operation	ASM Hook	ASM Callback
IMEI	`device_id_mod`	int get_device_id_mod(String fake_imei[])
Location updates	`on_location_changed_mod`	int on_location_changed_mod(int uid, Location loc)
Internet Access	`socket_connect`	int socket_connect(String family, String type, int uid)
Contacts Query	`contacts_query_mod`	int query_contacts_mod(ASMCursor c, String projection, ...)
Contacts Insert	`contacts_insert`	int contacts_insert(Uri uri, ContentValues values)
Contacts Delete	`contacts_delete`	int contacts_delete(Uri uri, String selection, String selectionArgs[], ...)
Contacts Update	`contacts_update`	int contacts_update(Uri uri, ContentValues values, String selection, ...)
Receive Broadcast	`resolve_broadcast_mod`	int resolve_broadcast_mod(List resolvedList, String resolvedtype, ...)

group. To enforce this mock permission, MockDroid modifies the `inet` runtime check in Android's Linux kernel. In the modified check, if the application belongs to the `mock_inet` group, a socket timeout error is returned, simulating an unavailable network server.

MockDroidASM. We implemented an ASM App version of MockDroid called MockDroidASM. In addition to ASM permissions for hook registration, MockDroidASM must register for the `PACKAGE_INSTALL` hook to receive the package name and the list of requested permissions when each new application is installed. A MockDroidASM GUI also allows the user to configure which permissions to gracefully revoke from an application (e.g., `INTERNET`, `READ_PHONE_STATE`).

Instead of using additional mock permissions, MockDroidASM registers for the modify version of ASM hooks that are triggered when an application attempts to access sensitive system components. Since MockDroidASM needs to modify values returned to apps, it requests the `REGISTER_ASM_MODIFY` permission, as described in Section 5.2.3.

Table 7 shows the most important hooks used by MockDroidASM. For example, the `device_id_mod` hook allows MockDroidASM to fake the IMEI number of the device. On the kernel-level, MockDroidASM registers for the `socket_connect` hook to receive a callback when an application tries to connect to a network server. If `INTERNET` is revoked by the user, the MockDroidASM returns deny to the ASM LSM, which returns a socket timeout error to the application.

5.3.2 *AppLock*

AppLock [63] is an application available on the Google Play Store. It allows users to protect the user interface components of applications with a password. Users set a password to access AppLock. They then selectively lock other third-party and system applications through AppLock's user interface. When the user tries to open a protected application, AppLock presents a password prompt, and the user must enter the correct password before the application can be used.

Original Implementation. AppLock requests install-time permissions for 1) getting the list of running apps, 2) overlaying its user interface over other applications, and 3) killing application processes. While AppLock does not require any modifications to Android's source code, it uses energy very inefficiently. It can also be circumvented using an Android Debug Bridge (ADB) shell (e.g., "`am force-stop com.domobile.applock`").

AppLock's *LockService* uses a busy loop to continuously query the Android operating system for the list of running applications while the screen is on. If the top application is protected by AppLock's policy, LockService overlays the current screen with a password prompt user interface. This interface stays on the screen, trapping all input until the correct password is entered. If the user decides to return from the lock screen without entering his password, AppLock kills the protected application. We have verified this execution via static analysis using ApkTool [282] as well as with another monitoring ASM App that registers for the `start_service` hook.

AppLockASM. We implemented an ASM App version of AppLock called AppLockASM. To provide the password-protected application functionality, AppLockASM simply registers for the `start_activity` hook. It then receives a callback whenever an Activity component is started. When this occurs, AppLockASM displays its own lock screen. If the user enters the correct password, the `start_activity` event is allowed. If the user decides not to enter a password, it is denied. Unlike AppLock, AppLockASM never starts the target Activity component without the correct password.

5.3.3 *App-specific Firewalling*

DroidWall [308] is a frontend for the Linux Netfilter packet filter, which enables Android users to establish firewall rules for individual applications on rooted devices. Using DroidWall, users can for example prevent applications from loading ads from remote servers or filter network traffic to known web tracking services.

Original Implementation. The original DroidWall app requires that the user first compromises the integrity of the Android operating system by rooting his device. This is required since Android by default does not provide any configuration interface for the kernel-level firewall. Once root access is acquired, the DroidWall app configures the kernel-level firewall using the *iptables* command line frontend. Rules are applied for individual applications by using the UIDs of applications as a filter argument.

DroidWallASM. We implemented the main functionality of DroidWall using ASM. Our DroidWall ASM App called DroidWallASM uses the `socket_connect` hook, which is called whenever an application opens a socket. It provides the socket type, family, and protocol as well as the destination address to DroidWallASM, which then at runtime decides whether or not to allow the connection attempt. In our current implementation, the firewall policy is specified by the user himself via a graphical configuration interface, which supports both black - and whitelisting of IP addresses. In an enterprise deployment, the firewall policy could instead be pushed to the device via a mobile device management solution.

Since filtering network traffic using IP addresses instead of hostnames is cumbersome, DroidWallASM allows users to alternatively define filter rules using the hostnames of

remote servers. To this end, we placed a hook into Android's network management daemon, which is denoted by *netd*: The `get_addr_info` hook is triggered whenever applications resolve hostnames to IP addresses. While DroidWallASM allows every `get_addr_info` protection event, it caches which IP addresses belong to which hostnames. Using this information, DroidWallASM can then selectively block or allow connection attempts to black- or whitelisted hostnames.

5.3.4 *Summary*

ASM considerably simplifies development of security modules such as AppLock, MockDroid and DroidWall. For example, the original AppLock app performs its functionality by starting a Service in an infinite loop, a design that is inefficient in terms of energy as well as latency. AppLockASM on the other hand needs to simply register for a callback with the ASM Framework. The AppLock implementation also prompts a lock screen after the app has already been started, and has to kill the app when the lock screen returns. This arbitrary killing of apps is prevented in the AppLockASM case, where the callback happens before the Activity is started, and the Activity starts only if the AppLockASM allows.

The original MockDroid implementation requires modifications to the PackageManagerService, and has to implement an entire parallel mock permission framework. This effort can be reduced by registering for a small number of ASM hooks, without having to modify system Services.

Similar to the original MockDroid implementation DroidWall requires the user to first violate the integrity of the underlying operating system by *rooting* his device. In contrast, our DroidWallASM implements the most important aspects of DroidWall, namely filtering network access on a per-application basis, using hooks provided by the ASM framework. We are further exploring how to provide a standardized and controlled interface to the Netfilter packet filter of the Linux kernel to ASM Apps in future versions of the ASM framework.

A general lesson learned from these case studies is that the ASM architecture enables developers to easily implement complex system-centric security enhancements without the need for third party support. This broadens the outreach of ASM, and encourages third-party developers to engage in the development of sophisticated security solutions for Android-based smart mobile devices.

5.4 EVALUATION

To evaluate our ASM framework in practice we implemented a prototype based on Android 4.4.1 AOSP and deployed it on a LG Nexus 4 (GSM) smartphone. In this section, we will analyze the impact of our framework on both performance and energy consumption.

Table 8: Performance evaluation - unmodified AOSP, ASM with no reference monitor, and ASM with a reference monitor app

	AOSP (ms)	ASM (ms)		Overhead (ms / %)	
Protection Event		**w/o ASM App**	**w/ ASM App**	**w/o ASM App**	**w/ ASM App**
Start Activity	19.03 ± 1.51	20.01 ± 1.39	22.74 ± 1.77	0.98(5.15%)	3.71(19.50%)
Start Service	3.89 ± 0.31	4.6 ± 0.41	8.42 ± 0.61	0.71(18.25%)	4.53(116.45%)
Send Broadcast	2.18 ± 0.24	4.48 ± 0.69	6.45 ± 0.55	2.30(105.50%)	4.27(196.71%)
Contacts Insert	121.41 ± 5.98	120.48 ± 5.25	135.39 ± 6.35	−0.93(−0.76%)	13.98(11.51%)
Contacts Query	17.41 ± 3.88	21.10 ± 3.13	29.50 ± 4.36	3.69(21.19%)	12.09(69.44%)
File Read	59.13 ± 1.97	62.27 ± 2.86	65.39 ± 2.93	3.14(5.31%)	6.26(10.59%)
File Write	57.68 ± 3.01	57.98 ± 2.76	59.03 ± 3.60	0.30(0.52%)	1.35(2.34%)
Socket Create	0.65 ± 0.086	0.79 ± 0.13	4.26 ± 0.56	0.14(21.54%)	3.61(555.38%)
Socket Connect	1.61 ± 0.21	1.65 ± 0.22	5.13 ± 0.32	0.04(2.48%)	3.52(218.63%)
Socket Bind	2.00 ± 0.17	1.93 ± 0.64	5.15 ± 0.34	−0.07(−3.5%)	3.15(157.50%)

5.4.1 *Performance Overhead*

To understand the performance implications of ASM, we micro benchmarked the most common ASM protection events for the modules presented in Section 5.4.1. We performed each experiment 50 times in three execution environments: 1) AOSP, 2) ASM with no ASM App, and 3) ASM with one ASM App. The ASM App only registers for the callback of the tested protection event; all other callbacks remain deactivated. Since we are only interested in the performance overhead caused by framework, our test callback immediately returns *allow*. Table 8 shows the mean results with the 95% confidence intervals.

Lifecycle Protection Events. To test lifecycle protection events (i.e., start Activity, start Service, and send Broadcast), we created an Intent message and added a byte array as its data payload (i.e., extras Bundle). Each test type registered for the modify version of the ASM hook. We sent the Intent for the respective type, pausing for five seconds between consecutive executions. Potential areas of overhead for using the hook include: 1) cost of establishing two additional IPCs, 2) marshalling and unmarshalling this data across the two IPCs, 3) ASM copying the extras Bundle when sending it to the ASM App, and 4) setting the returned Bundle back to the original Intent. To estimate worst case performance, we chose a very large array (4 kB) and registered our test ASM for modify data hooks. This worst case overhead, though relatively high, is not noticeable by the user due to its low absolute value. Additionally, most applications use files to share very large data values. We note that while send Broadcast has a high overhead percentage, the wall clock overhead is in the order of milliseconds, which is negligible overhead for Broadcast Intents.

ContentProvider Protection Events. Micro benchmarks for ContentProviders were performed on the ContactsProvider. For this experiment, our ASM App registers for the `contacts_insert` callback. It proceeds to insert a new contact (first and last name) into the ContactsProvider. The overhead observed is 11.51% and negligible in terms of its absolute value. We then registered for the `contacts_query_mod` hook, and performed a

query on the same contact. Query has a greater overhead, which is attributable to marshalling/unmarshalling the data between the two IPC calls, and serialization of the Cursor object into a parcelable. A major cause of this overhead is also that the Cursor is not populated when the query result is returned to the calling application, but is instead filled as and when the application uses it to retrieved values.

File Access Protection Events. File micro benchmarks tested the `file_permission` hook, which uses an upcall from the kernel. To test file access performance, our test app performs an access (read/write) on a 5 MB file. We pause for a second between successive executions. For writes, we do not see considerable overhead as the file is written in one shot to disk. Reads used a 16 kB buffer and the default 1 ms expiration time for caching access control decisions, as discussed in Section 5.2.4.

Socket Protection Events. For socket operations, we tested the performance overhead for creating, binding and connecting to an IPv6 socket. Our test ASM App registered for the `socket_create`, `socket_bind`, and `socket_connect` callbacks. The absolute overhead is mainly caused by the callback to the userspace, and is a constant overhead for socket operations.

5.4.2 *Energy Consumption*

Energy consumption is a growing concern for smart mobile devices. To measure ASM's impact on energy consumption, we perform energy measurements in same three test environments as performance: 1) AOSP, 2) ASM with no ASM App, 3) ASM with one ASM App. The ASM App registers for all the hooks from the performance experiments. We use the Trepn profiler 4.1 [195] provided by Qualcomm to perform power measurements. Trepn uses an interface exposed by the Linux kernel to the power management IC used on the System on a Chip to measure energy consumption, a feature that is supported on a limited set of devices, including the LG Nexus 4. Trepn samples power consumption measurements every 100 ms. Average values are shown in the Table 9.

We monitor system energy consumption while running the test applications from Section 5.4.1. When the hooks are deactivated, we measured an energy consumption overhead of about 3.34%. Our ASM App used for the performance and energy consumption experiments measured an overhead of about 9.33%. This overhead is caused by the active authorization hooks in the relevant OS components and kernel, as well as the communication between the authorization hooks, the ASM Bridge, and the ASM App.

It should be noted that performing accurate energy consumption measurements on smartphones is challenging. While we consider the individual measurements to be accurate, we acknowledge that the low sampling rate used by the Trepn profiler is problematic. However, each individual experiment is performed 50 times, therefore we believe Trepn's measurements to at least provide a rough estimate of the energy consumption overhead introduced by ASM.

Table 9: Energy consumption overhead of ASM

Environment	**Average Power Consumption (in mW)**	**Overhead (%)**
AOSP	670.42	-
ASM w/o ASM App	692.83	3.34
ASM w/ ASM App	732.98	9.33

5.5 CONCLUSION

In this chapter, we presented the Android Security Modules framework, a programmable interface for extending Android's security architecture. While similar reference monitor interfaces have been proposed for Linux and TrustedBSD, ASM is novel in how it addresses the semantically rich OS APIs provided by new operating systems for smart mobile devices such as Android.

ASM promotes the creation of novel security enhancements to Android without restricting OS consumers (e.g., consumers, enterprise, government) to specific policy languages (e.g., type enforcement). In its current state, ASM allows researchers with the ability to recompile Android to rapidly prototype novel reference monitors without needing to consider authorization hook placement. If ASM is adopted into the AOSP source code, it potentially allows researchers and enterprise IT to add new reference monitors to production Android devices without requiring root access or replacing the firmware, a significant limitation of existing bring-your-own-device solutions.

6 PRACTICAL USE CASES

In this chapter, we will discuss a set of practical use-cases for system-centric access control. Section 6.1 demonstrates how the FlaskDroid and ASM architectures presented in Chapters 4 and 5 can improve both security and user experience by enforcing context-aware access control rules. Section 6.2 proceeds to show the value of deploying system-centric access control architectures for application behavior analysis. In Section 6.3 we discuss system-centric access control for security domain isolation based on our TrustDroid [33] architecture, focusing on its commercial adoption in the award-winning BizzTrust solution [229]. Finally, in Section 6.4 we present an alternative application-layer access control architecture which addresses situations where the deployment of system-centric solutions is currently not feasible.

6.1 CONTEXT-AWARE ACCESS CONTROL

Smart mobile devices, such as smartphones and tablets, have become omnipresent personal and enterprise assistants. They serve as a primary communication mechanisms, manage privacy-sensitive data and host a wide variety of personal and enterprise applications. To increase usability, these devices are equipped with a substantial number of sensors, which enable installed applications to dynamically react to their environment and to serve contextual information whenever needed.

The availability of high-quality sensor information both poses challenges and provides opportunities: On one hand, sensor information is a sensitive resource which needs to be protected from illegitimate access: As discussed in Section 3.3.3 related work has demonstrated a variety of attacks using on-device sensors to derive sensitive information while the device is used in sensitive environments [218, 288, 251, 159]. On the other hand, contextual sensor information can be a useful tool to derive security requirements for the current usage context. Consider for example an enterprise security policy which dictates that access to the on-device camera and microphone must be restricted while the device is located on enterprise premises, or devices automatically balancing usability and security requirements depending on the current risk of device misuse or theft.

Enforcing access control rules depending on the usage context and environment is an active area of research, and related work has discussed a variety of corresponding mechanisms. The proposed approaches range from abstract context-aware role-based access control models [46, 50], where roles are triggered based on contextual information, to concrete context-aware access control frameworks targeting mobile devices, where context parameters are considered as additional conditions during policy decisions [212, 17, 45, 23, 180]. However, these approaches all use inflexible and predefined policies. Consider for example popular geofencing mechanisms, which are actively used by mobile device management software to restrict access to sensitive information stored on or accessed by the device [98, 226].

Setting up geofences is an elaborate and error-prone process, which does not adequately address the highly dynamic nature of usage contexts: Geofencing only considers the geographic location of the device, but not dynamic threats present in the environment, such as unfamiliar people and their devices.

In this section, we present ConXsense, an alternative approach which enforces access control rules depending on dynamically established usage contexts. Our solution derives these usage contexts automatically using machine learning techniques and takes both the geolocation as well as the familiarity of surrounding devices into account. We integrate ConXsense with both our FlaskDroid and Android Security Modules access control architectures to protect sensitive sensor data from illegitimate access. We further show how contextual information can be used to improve usability while simultaneously protecting devices against misuse by unauthorized persons.

Contribution. To summarize, our main contributions are as follows:

- **Context-Aware Access Control.** We present a novel context-aware access control framework for smart mobile devices. Our architecture bases its access control decisions on automatically generated context classifications derived via machine learning techniques.
- **Protection against Sensory Malware.** We demonstrate how context classifications can be used to protect users from sensory malware.
- **Usable Device Lock.** We show the value of contextual information for balancing usability and security aspects of smart mobile devices.

Remark. The results presented in this section were achieved in collaboration with Markus Miettinen, Wiebke Kronz, Ahmad-Reza Sadeghi and N. Asokan. High-level idea and main design were proposed by Markus Miettinen, who also implemented all aspects concerning context profiling, classification and evaluation. These aspects are out of scope of this dissertation and will only briefly be discussed. Wiebke Kronz contributed a social study concerning contextual security. All aspects concerning access control design, implementation and evaluation are due to the author of this dissertation. N. Asokan and Ahmad-Reza Sadeghi were both involved in fruitful discussions and provided feedback which improved the quality of our publication [162].

6.1.1 *Scope*

To define the scope and goals of our work we first performed a user survey in April 2013 [162] to identify user concerns regarding contextual sensor information. Our survey identified two main concerns, which ConXsense aims to address:

The first concern is unauthorized disclosure of security- and privacy-sensitive data. As discussed previously in Section 3.3.3 related work has demonstrated a variety of attacks using sensory malware to derive sensitive user data. For example, the on-device camera allows sensory malware to observe the user's surroundings [288, 251], while the microphone

can be used to recover sensitive credit card information from conversations [218]. Further, it has been shown that it is feasible to extract user credentials entered on a nearby computer keyboard by evaluating acceleration sensor information on a smartphone or tablet [159].

The second concern is device misuse. Related work has identified that users are not willing to use complicated idle screen locks to prevent such misuse [38, 227] due to usability issues. Consider for example an idle lock screen which requires the user to enter a long and complicated password. Gupta et al. [108] proposed to address this concern by changing the lockscreen properties based on the current usage context. For example, the operating system could enforce that the user has to manually unlock his device every time it is used in an "unsafe" context with high risk of device misuse. In contrast, in order to improve usability it is reasonable to relax this policy in "safe" contexts.

Requirements. To address these concerns our ConXsense architecture needs to fulfill the following set of requirements:

- **Mitigation of Sensory Malware.** ConXsense prevents sensory malware from gaining access to sensitive data by restricting access to sensitive sensor information to a user's trusted applications while the device is used within sensitive environments.
- **Protection against Device Misuse.** ConXsense restricts access to sensitive data and functionality in situations where it is likely for an adversary to get physical access to the device. While doing so, ConXsense does not negatively affect usability.

6.1.2 *Design*

High-Level Idea. Our ConXsense architecture enforces context-dependent access control on access to sensor information. It further dynamically configures lockscreen policies depending on the security properties of the current environment. Our architecture uses supervised learning to derive these contextual security properties. It then enforces corresponding access control decisions with the help of a system-centric access control layer, such as FlaskDroid or Android Security Modules (see Chapters 4 and 5).

Our architecture is depicted in Figure 13 and comprises four main components. The Context Data Collector is an application which is installed on a user's device and collects contextual observations as well as ground truth data. These observations concern sensor information, such as the geolocation as well as surrounding devices discovered via WiFi and Bluetooth sensing. Our Context Profiler then generates corresponding feature vectors from these observations. Finally, the Context Classifier uses supervised learning based on the generated context feature vectors and collected ground truth data to train a context model and classifies the current context. Based on this classification the Access Control Layer enforces context-dependent access control rules. In the following, we will describe these components in more detail. Note however that this dissertation focuses on access control aspects of ConXsense. We thus will only present a brief description of our work on context detection and classification and refer to our publication [162] for a more detailed discussion.

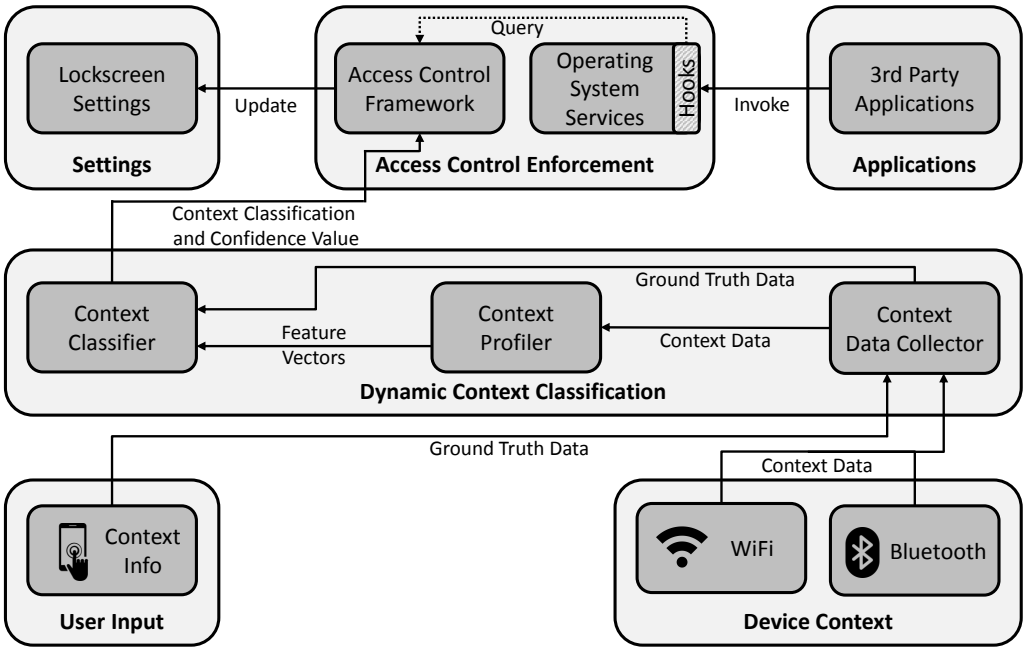


Figure 13: ConXsense framework architecture

Context Data Collector. Our Context Data Collector collects context information. This information comprises sensory data, such as the GPS-derived geolocation of the device, nearby Bluetooth devices as well as WiFi access points. The Context Data Collector further collects ground truth information from the user, which is required to train our context models using supervised learning. To this end, the user is periodically requested to categorize the current context as "work", "private" and "public". These distinct categories have been established by our social study mentioned in Section 6.1.1. Further, our study shows that users experience these categories to be either "safe" and "unsafe", depending on the perceived risk for device theft and misuse, and our Context Data Collector requests this information from the user as well.

Context Profiler. In our architecture the Context Profiler evaluates context observations collected by the Context Data Collector. To this end, the Context Profiler first evaluates both GPS geolocation as well as surrounding WiFi access points to identify *Contexts of Interest (CoIs)* where a user spends a significant amount of time.

Once CoIs have been established the Context Profiler evaluates both the *location context* and the *social context* of the device. The location context is defined as the CoIs the user is visiting at a given point in time based on the current WiFi observations and GPS geolocation of the device. The social context, on the other hand, is based on observations of surrounding Bluetooth devices. The main idea is to categorize nearby devices as *familiar* when they are encountered regularly and for a long period of time. For example, a spouse's device which is detected regularly at home will be categorized as a familiar device, whereas devices encountered while visiting a public space, such as a supermarket, would not be categorized as familiar.

Finally, the Context Profiler calculates feature vectors based on the frequency and duration of CoI encounters and detected familiar devices. These features are labeled according to the feedback provided by the user to the Context Data Collector and used as training data for the Context Classifier.

Context Classifier. Our Context Classifier uses the established labeled feature vectors to train machine-learning based classifiers by means of supervised learning. Once the classifiers are trained, they classify new context observations as "work", "private" or "public". They further estimate the risk for device misuse and apply the "safe" or "unsafe" label accordingly.

Access Control Framework. To enforce context-dependent access control rules we designed two variants of our access control layer. Our initial solution discussed in our publication [162] is based on FlaskDroid (see Chapter 4) and uses type enforcement. We later adapted our solution to the more recent Android Security Modules access control architecture (see Chapter 5), which provides a programmable interface for implementing reference monitors. Based on the classification of the current context our Access Control Layer restricts access to on-device sensors by third-party applications. In addition, we modified the Android operating system to allow the Access Control Layer to influence the Lockscreen behavior at runtime.

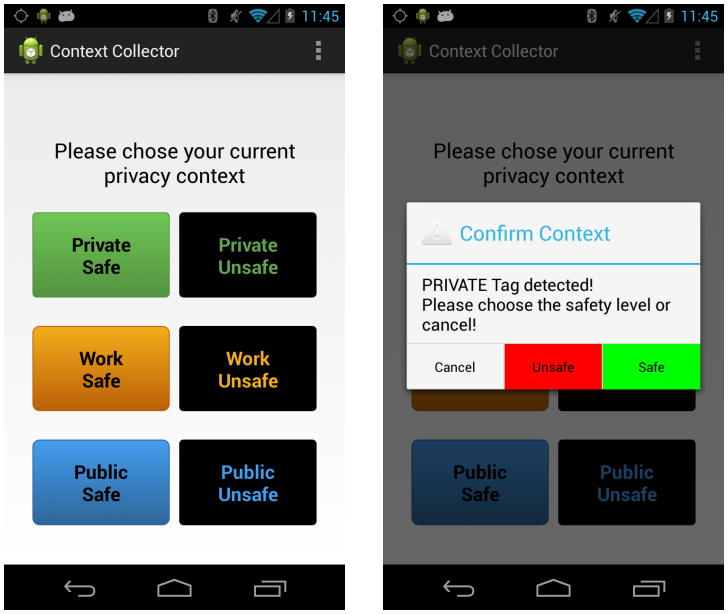
6.1.3 *Implementation*

Context Data Collector. We implemented a Context Data Collector app for Android which uses a background Service to collect context information in intervals of 60 seconds, which is a tradeoff between the amount of collected sensor data and the energy consumption of the device. Our Context Data Collector observes location data, WiFi access points as well as nearby Bluetooth devices. In addition, users provide ground truth data for training our context models using the Context Data Collector user interface as well as preprogrammed NFC tags provided to the participants (see Figure 14). Users were instructed to spontaneously provide ground truth data, and our Context Data Collector reminded them to do so in case no feedback had been provided during a two hour period. Context and ground truth data were opportunistically uploaded to a remote server via HTTPS whenever stable WiFi network connectivity was available.

Context Profiler. Our Context Profiler evaluates the collected context information and generates feature vectors using *Python*, *bash scripts* and *awk*. It generates GPS and WiFi CoIs for every user and evaluates Bluetooth observations to identify familiar devices.

Context Classifier. The Context Classifier was implemented based on the Weka data mining suite [110], which provides implementations for the necessary machine learning classifiers used during our evaluation of ConXsense (see Section 6.1.4).

Access Control Enforcement. We integrated the FlaskDroid and Android Security Modules access control architectures with our framework to enforce context-aware access control rules. To do so, we first implemented a ContextProvider for the FlaskDroid



(a) Feedback using Context Feedback Buttons

(b) Feedback using Context NFC Tags.

Figure 14: Android Context Data Collector app

architecture and defined a corresponding access control policy, which we deployed on a FlaskDroid-enabled Samsung Galaxy Nexus smartphone running Android 4.0.4. We later ported the core logic of the ContextProvider to the Android Security Modules architecture and deployed the resulting ASM App on a LG Nexus 4 smartphone running Android version 4.4.1 and our ASM framework version r1.

FlaskDroid based Instantiation. Our FlaskDroid architecture (see Chapter 4) extends *Security Enhanced Android (SEAndroid)* [232] with fine-grained type enforcement on the middleware- and application-layer. In FlaskDroid, Android middleware components that provide access to sensitive resources, such as the SensorService which provides access to sensor information, act as Userspace Object Managers (USOMs). These USOMs enforce access control on resources they manage. More specifically, USOMs control *operations* of *subjects* (i.e., apps) on *objects* (e.g., sensor data) based on *types* assigned to both *subjects* and *objects*.

At boot time, FlaskDroid's Userspace Security Server parses an Access Control Policy and proceeds to assign app types (e.g., *trusted* or *untrusted*) to all installed apps based on application metadata (e.g., package name or developer signature). Apps installed by the user are assigned types during installation. Whenever apps request access to a USOM, for example the SensorService to query the device's sensors or the CameraService to take pictures, the USOM queries the Userspace Security Server, which is part of Android's SystemServer, for access control decisions. As noted in Section 4.2.5 FlaskDroid supports conditional access control rules by means of ContextProviders, which evaluate the current context and enable or disable rules at runtime.

To meet our goals we thus first implemented a ConXsense ContextProvider. It uses the context classification information and confidence values provided by the Context Classifier to activate or deactivate conditional rules at runtime and to influence the Lockscreen behavior. The ContextProvider can be tuned with individual user-, use-case and sensor-specific thresholds for the expected confidence values. These thresholds could be set, for example, by specifying a desired maximum false positive rate and adjusting the confidence threshold accordingly based on the observed historical performance of the Context Classifier. Further, access to more sensitive context sensors like GPS could require a higher prediction confidence than less sensitive sensors like the magnetometer.

To mitigate, respectively reduce, the effects of sensory malware (e.g., *Placeraider* [251] or SoundComber [218]) access control on the sensors of a device is required. For example, *Placeraider* uses the device's camera and the acceleration sensor to covertly construct 3D images of the surroundings of the user. We designed a type enforcement policy which filters queries to FlaskDroid's CameraService USOM. Furthermore, ConXsense uses FlaskDroid to filter acceleration sensor events delivered to SensorEventListeners registered by apps using FlaskDroid's SensorService USOM. Similarly, the combination of ConXsense and FlaskDroid can address other variants of sensory malware, such as *Soundcomber* [218], by identifying the relevant Android APIs, instrumenting them as USOMs and enforcing corresponding access control policies.

Finally, to allow for changes in Android's Lockscreen policy based on the current risk for device misuse we use the ConXsense ContextProvider to configure Android's Lockscreen dynamically at runtime. We modified Android's Settings component to be notified by our ContextProvider about changes in the current risk for device misuse via Broadcast Intents. We further modified Android's LockPatternKeyguardView, which is a Java class responsible for displaying the Lockscreen, to query the Settings component for context information. While the device is used in a context with *low* risk for device misuse the LockPatternKeyguardView class automatically dismisses the Lockscreen. Whenever the device is rebooted or the risk for device misuse changes to *high*, a low-watermark mechanism ensures that the Lockscreen is always displayed regardless of the current risk for device misuse. This mechanism is required to prevent an attacker from bypassing the Lockscreen by changing the context, emulating a low-risk context or rebooting the device. In addition, to mitigate the effect of sensory malware which uses the acceleration sensor as a side channel to derive user credentials (e.g., Lockscreen PIN or password) [290, 183, 37] we use FlaskDroid's SensorService hooks to block access to the acceleration sensor by 3rd-party apps while the Lockscreen is displayed.

Android Security Modules based Instantiation. We ported our initial FlaskDroid-based design to the more recent ASM architecture, which enforces access control decisions using hooks integrated into security- and privacy-sensitive operating system components (see Chapter 5). ASM exposes these hooks to application-layer security modules, denoted ASM Apps. Our ConXsense ASM App registers itself with the ASM architecture and uses SensorService hooks to control access to on-device sensors as well as CameraService hooks to control access to the camera. More precisely, our ConXsense ASM App registers for the `camera`, `on_location_changed_mod` and `sensors` hooks. We further integrated our previ-

ously described changes within Android's Settings component into ASM to influence the Lockscreen behavior at runtime depending on the context.

6.1.4 *Evaluation*

To evaluate our approach the Context Data Collector was deployed on a group of 15 test users' Android devices. We collected context and ground truth data for a total period of 68 days, achieving 844 distinct user days and 3575 labeled data points. The collected data was evaluated by the Context Profiler, which generated personal context profiles and context feature vectors.

Our Context Classifier applied three different machine learning algorithms to the labeled feature vectors established by the Context Profiler. We experimented with a k-nearest neighbors (kNN) classifier, a Naïve Bayes classifier as well as a classifier based on Random Forests. The Context Classifier's task was to predict situations, in which the restrictive default protection mechanisms (short idle lock timeout, no sensor access for untrusted applications) could be relaxed. In our use case, the idle lock timeout could be extended when the device was used in a "safe" context, whereas the sensor access for untrusted applications could be enabled in "public" environments.

To evaluate the performance of the selected classifiers we considered all users which uploaded at least five labeled data points for every context class. Our results show that all selected classifiers achieve a reasonable performance. However, since this dissertation focuses on access control aspects of ConXsense the following paragraphs will only provide a qualitative assessment of our results. We refer to our research paper [162] for a more detailed analysis and discussion.

Protecting against Device Misuse. In this use-case the classifier's objective was to identify scenarios with low risk of device misuse in which the Lockscreen policy could potentially be relaxed. In our experiments all classifiers were able to achieve a true positive rate of 70% while maintaining a false positive rate of 10%. Accordingly, our modified Android Lockscreen would have applied a relaxed policy 70% of the times the device was used in "safe" contexts, and would have incorrectly applied this relaxed policy 10% of the times while the device was used in "unsafe" environments. While improving this false positive rate is an open challenge, even in its current state our solution could improve security for users who would otherwise disable the Lockscreen entirely.

To test our implementation of the context-aware device Lockscreen we modified the Android operating system to periodically wake the device from sleep and switch on the screen. We then installed a synthetic malware, which registers SensorEventListeners in Android's SensorService to be notified of acceleration sensor readings. By logging and analyzing the Lockscreen behavior, context information and sensor readings we verified that the Lockscreen was only automatically dismissed in valid situations and that our synthetic malware did not receive any sensor readings while the Lockscreen was active.

Protecting against Sensory Malware. In this use case the goal of the Context Classifier was to identify contexts with low privacy exposure where the access control policy on sensors could be relaxed to grant third-party applications access to all on-device sensors.

Our experiments show that the Random Forest and kNN classifiers were able to achieve a true positive rate of 70% while maintaining an acceptable 2 - 3.5% false positive rate. Thus, only 2 - 3.5% of the times the device was used in a context with high privacy exposure sensory malware would have been granted access to sensitive sensor information. In contrast, 70% of the time the device was used in contexts with low privacy exposure third-party applications would rightly have been granted access to sensor information. To address the remaining 30% we suggest a manual override mechanism which instructs the access control layer to temporarily ignore the decision of the Context Classifier. In fact, the revocable permission architecture introduced in recent versions of the Android operating system (see Section 2.3.2) is a prime candidate to implement such a solution. This override would further serve as another labeled data point for the training of our context models and thus could potentially improve the accuracy of our Context Classifier over time.

We tested our integration of the FlaskDroid and ASM access control architectures into ConXsense using a slightly modified version of the *PlaceRaider* malware generously provided to us by its authors[1]. By installing the malware on our device and logging context information and access control decisions we verified that both FlaskDroid and ASM successfully filtered all data delivered from Android's SensorService and CameraService components to the *untrusted PlaceRaider* app when the risk for privacy exposure was *high*, thus rendering the attack futile. We further verified that *trusted* apps could still use the sensors and the camera. No additional false positives or false negatives emerged during the evaluation of the Access Control Layer, which is not surprising since it merely enforces context-dependent access control rules.

6.1.5 *Conclusion*

In this section, we discussed the ConXsense architecture, which implements context-aware access control based on our previously introduced FlaskDroid and Android Security Modules frameworks. ConXsense uses sensor information to automatically classify the current usage context of the device. Based on this information, our architecture dynamically reconfigures access control rules to better protect security- and privacy-sensitive data in high-risk situations. Our evaluation shows that our approach can effectively address real-world security concerns of users, such as malware accessing sensitive sensor information and device misuse.

To extend our work on context-aware access control the following aspects should be considered: First, an on-device implementation of the Context Profiler and Context Classifier to augment our current offline implementation. Second, while our user study identified a set of practical use-cases for context-aware access control usability aspects of our ConXsense implementation itself should be evaluated as well. Finally we envision the integration of additional contextual information, ranging from sensor data to observations about the user's interaction with the mobile device and deployed applications, to improve the performance of our context model.

[1]The sample we received is incompatible with recent versions of Android.

6.2 ACCESS CONTROL FOR APPLICATION BEHAVIOR ANALYSIS

Smart mobile devices host a vast number of third-party applications of varying quality and trustworthiness. These applications access, store and process security- and privacy-sensitive data, ranging from personal contacts, location information to high-profile enterprise assets, which makes these devices valuable targets for attacks. Related work has shown that Android's default permission-based access control model is susceptible to application-layer privilege escalation attacks, ranging from insufficiently protected system settings [81] to accessing the Internet [152] or sending SMS [52] without holding corresponding permissions.

On one hand, system-centric access control architectures can mitigate such attacks, but require carefully designed use-case specific security policies (see Section 4.4.2). On the other hand, both static and dynamic program analysis promise to proactively detect such attacks (see Section 3.4.1 and 3.4.2), but either do not adequately address native and highly obfuscated code, or are susceptible to malware using logic bombs to avoid early detection [201, 271, 139].

This inability to proactively and reliably detect application-layer privilege escalation attacks mandates tools for long-term observation and analysis of application behavior. In this section, we present DroidAuditor, an application behavior analysis toolkit targeting application-layer privilege escalation attacks. DroidAuditor adopts our Android Security Modules access control architecture to observe application behavior at all layers of the Android operating system. Our solution organizes these observations in a *behavior graph* and generates an interactive visualization. It further allows security analysts to query this graph for suspicious patterns using a graph query language.

Contribution. To summarize, our main contributions are as follows:

- **Application Behavior Observation using Android Security Modules.** We show that sophisticated access control frameworks, such as the Android Security Modules framework introduced in Chapter 5, are a valid basis for application behavior analysis.
- **Interactive Visualization and Intuitive Analysis.** We present the design and implementation of DroidAuditor, a solution for application behavior analysis based on interactive behavior graphs.
- **Extensive Evaluation.** We evaluate our architecture by analyzing application-layer privilege escalation attacks as well as malicious spyware apps.

Remark. The results presented in this section were achieved in collaboration with Marco Negro and Praveen Kumar Pendyala. Main idea and high-level design are due to the author. Marco Negro provided the initial implementation, which was refined by Praveen Kumar Pendyala and the author. Evaluation was performed by both Praveen Pendyala and the author. The author was tasked with writing our publication [120] based on initial input by Marco Negro. Ahmad-Reza Sadeghi provided feedback which improved the quality of our publication.

6.2.1 *Scope*

The main goals of DroidAuditor are the systematic monitoring of application behavior and the detection as well as analysis of potential application-layer privilege escalation attacks. In accordance with our adversary model presented in Section 3.1 we assume the adversary to be capable of deploying one or more malicious applications on a target user's device, for example via social engineering or by gaining temporary physical access to the device. We further assume that these applications perform application-layer privilege escalation attacks to expand their initial set of privileges.

Requirements. Given our adversary model and goal, we formulate the following requirements for DroidAuditor:

- **Application Behavior Observation.** DroidAuditor captures a target application's privacy- and security-sensitive behavior in a suitable data structure.
- **Application Behavior Analysis.** DroidAuditor enables a security analyst to discover and analyze potential application-layer privilege escalation attacks, such as confused deputy and collusion attacks.
- **Visual Representation of Application Behavior.** DroidAuditor generates an intuitive visual representation of application behavior and potential application-layer privilege escalation attacks.
- **Extensibility.** DroidAuditor is modular and allows a query-driven analysis of observed application behavior to enable further use-cases.

6.2.2 *Design*

The high-level idea of DroidAuditor is to observe application behavior using the system-centric Android Security Modules (ASM) access control framework presented in Chapter 5. DroidAuditor stores these observations in a *behavior graph*, where vertices represent applications and resources, and edges represent data- or control flows.

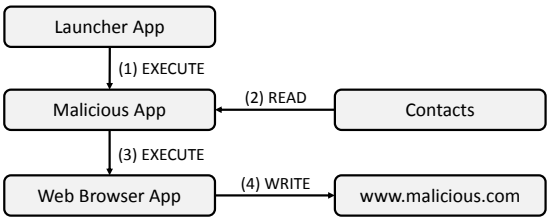


Figure 15: Example confused deputy attack, where a malicious app deputizes the web browser to exfiltrate sensitive contacts information.

Consider the following confused deputy attack: A malicious Android application holds the `READ_CONTACTS` permission and abuses the web browser to exfiltrate sensitive contacts information to a remote server without holding the `INTERNET` permission. Figure 15 shows

this attack as a behavior graph: Upon start (Step 1) the malicious app reads sensitive data from the ContactsProvider (Step 2). It then starts the web browser via an Intent (Step 3) and instructs it to exfiltrate contacts data on its behalf. The web browser opens a network socket to a remote server and uploads the collected contacts information (Step 4) to a server controlled by the adversary.

DroidAuditor generates such behavior graphs using three main components (see Figure 16). Whenever Android applications access security- or privacy-critical resources (Steps 1 - 4) the on-device DroidAuditor ASM App is notified by the ASM framework (Step A). The DroidAuditor ASM App forwards these notifications, denoted *protection events*, to the remote DroidAuditor Database via an authenticated and encrypted channel (Step B), where they are stored in the behavior graph. Finally, security analysts can interact with the behavior graph using the DroidAuditor Client (Step C).

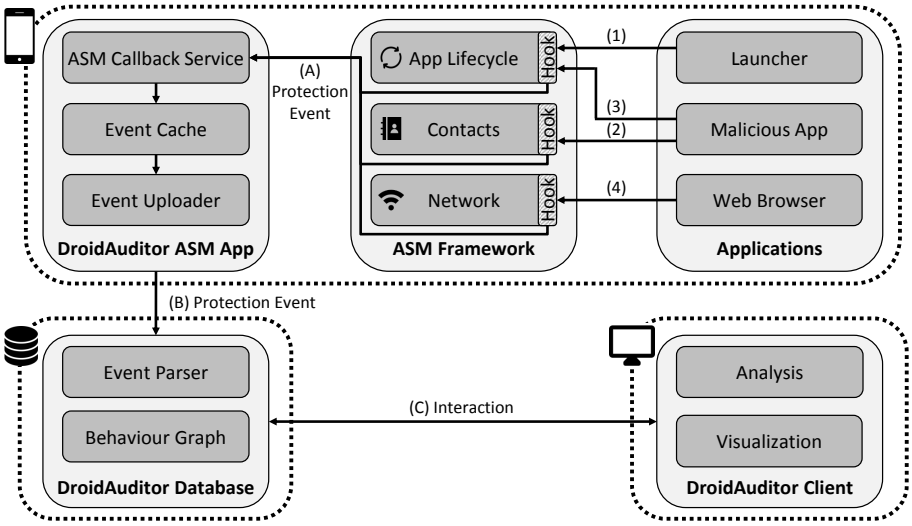


Figure 16: DroidAuditor framework architecture

DroidAuditor ASM App. The Android Security Modules framework places hooks in all security- and privacy-sensitive kernel-, middleware- and application-layer operating-system components. These hooks generate aforementioned protection events, which the ASM framework forwards to all installed ASM Apps. Each ASM App can then decide whether to allow or deny the corresponding operations (see Section 5.2). Our DroidAuditor ASM App however does not enforce any access control rules, but collects protection events to obtain a *global view* of all privacy- and security-critical operations performed by *all* applications. Accordingly, it allows every access control query and periodically uploads cached protection events to the remote DroidAuditor Database.

DroidAuditor Database. The DroidAuditor Database stores security- and privacy-sensitive protection events for offline analysis. It parses events uploaded by the DroidAuditor ASM App and generates the *behavior graph* $G = \langle V, E \rangle$: The vertex set $V = A \cup R$ is composed of two subsets A and R, which represent applications A and resources R. For each application vertex $a \in A$ the DroidAuditor Database stores an identifier as well as additional metadata, for instance the permissions the application holds. Each resource

vertex $r \in R$ models a security- or privacy-sensitive operating system resource. Important examples are Android's ContactsProvider, LocationManagerService or CameraService, as well as files and network sockets.

Every edge $e \in E$ is directional and describes a data- or control flow between two vertices $v_i, v_j \in V$. Each edge contains descriptive metadata, such as the time and date a flow was observed. Edges are grouped into categories, which model Android component interaction as well as file system and network operations (`CREATE`, `READ`, `WRITE`, `UPDATE`, `DELETE`, `EXECUTE`).

DroidAuditor Client. The DroidAuditor Client is a desktop application that interacts in real-time with the DroidAuditor Database. Its purpose is twofold:

First, the DroidAuditor Client generates an interactive visual representation of the behavior graph, which allows security analysts to intuitively gain an understanding of a target application's runtime behavior. Analysts can inspect the type and metadata for each vertex and edge as well as observe changes in the graph over time.

Second, the DroidAuditor Client allows analysts to query the behavior graph for specific patterns using the Cypher query language [171]. Listing 16 demonstrates how to query the behavior graph for signs of the previously described confused deputy attack, where a malicious app deputizes the web browser to exfiltrate sensitive contacts information. The depicted query identifies subgraphs starting with apps reading the Contacts resource (Lines 1-2). We only consider applications which then execute the Android web browser (Line 3) and do not hold the `INTERNET` permission (Line 5). Finally, this query expects the web browser to write data to a network socket (Line 4). Matching subgraphs are highlighted within the DroidAuditor Client.

Listing 16: Cypher query to detect the confused deputy attack

```
1 MATCH confuseddeputy = (contacts:Resource {type:'contacts'})
2 - [event1:READ] -> (app1:App {systemApp:false})
3 - [event2:EXECUTE] -> (app2:App {package:'com.android.browser'})
4 - [event3:WRITE] -> (socket:Resource {type:'socket'})
5 WHERE NOT 'internet' IN app1.permissions
```

6.2.3 *Implementation*

We implemented a prototype of the DroidAuditor architecture using the Java programming language. In the following, we will describe important implementation aspects of the DroidAuditor ASM App, Database and Client.

DroidAuditor ASM App. The DroidAuditor ASM App has been developed using revision r2 of the Android Security Modules Framework, which is based on Android 4.4.4. Our implementation follows the ASM App design discussed in Section 5.2.1, but never denies any access control queries. Instead, it merely logs every protection event to gain a global view of all privacy- and security-sensitive operations performed by all applications. The DroidAuditor ASM App caches these events in memory and serializes them to persistent on-device storage in regular intervals in JSON format. Whenever adequate and stable WiFi

or cellular network connectivity is available, it uploads cached events to the DroidAuditor Database.

DroidAuditor Database. We implemented the DroidAuditor Database using version 2.1.6 of the sophisticated Neo4j [170] graph database engine. Graph database engines efficiently store relationships between entities using graphs, and we selected Neo4j mainly for two reasons: First, it is a stable and mature database engine which has been shown to scale well and to provide adequate performance, even in the high performance computing field [62]. Second, Neo4j provides comprehensive developer tools, helper libraries, and documentation.

While in our current implementation the DroidAuditor ASM App and the DroidAuditor Database communicate using the Kryonet [245] network communication library over TCP, DroidAuditor is generic and supports other communication mechanisms as well, such as the upload of events via HTTPS. Uploaded protection events are interpreted using an event parser, which generates relevant application and resource vertices as well as edges representing relationships between them. For network-related protection events, such as those generated whenever applications connect to remote network servers, the event parser can additionally perform reverse DNS lookups to determine potential hostnames of these servers.

DroidAuditor Client. The DroidAuditor Client provides a generic runtime environment for graph visualization and analysis plugins. It is implemented as a Java Swing application and communicates with local or remote DroidAuditor Databases. We implemented the following plugins for the DroidAuditor Client:

Visualization Plugin. The visualization plugin generates an interactive visual representation of the behavior graph using the GraphStream library [254], which provides native support for Neo4j. Our implementation allows a security analyst to inspect individual vertices and edge meta-data as well as to observe changes in the behavior graph over time.

Analysis Plugin. The analysis plugin serves as a frontend for Neo4j's *Cypher* graph query language [171]. It enables a security analyst to interactively query the behavior graph for application and resource vertices, their properties and relationships, as well as subgraphs which could represent specific attacks. Whenever a subgraph matches a given Cypher query, it is highlighted in the behavior graph using the visualization plugin.

6.2.4 *Evaluation*

DroidAuditor inherits the performance overhead introduced by the underlying Android Security Modules framework, which has been described in detail in Section 5.4.1. In this section we thus primarily focus on DroidAuditor's effectiveness to analyze malicious application behavior. To this end, we deployed our previously described implementation of the DroidAuditor framework on a LG Nexus 4 device. We then installed applications which implement confused deputy and collusion attacks as well as malicious spyware applications on the device and analyzed their behavior. This section further discusses to what extend our architecture can be used to detect operating system level privilege escalation attacks.

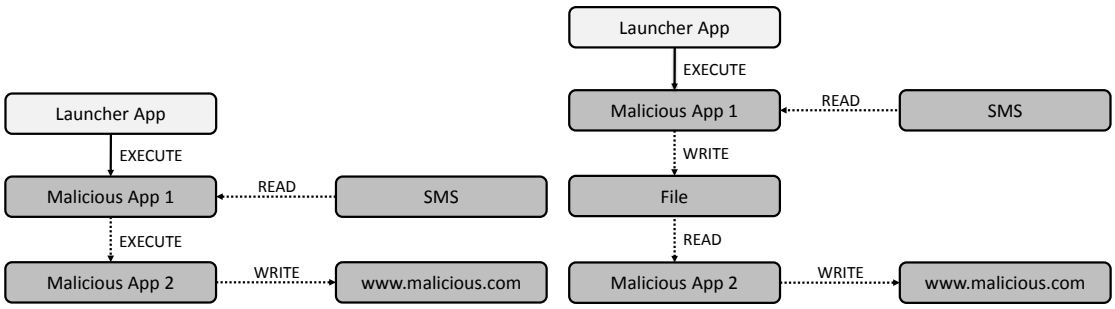
Listing 17: Cypher query to detect the collusion attack depicted in Figure 17a

```
1 MATCH collusion1 = (sms:Resource {type:'sms'})
2 - [event1:READ] -> (app1:App {systemApp:false})
3 - [event2:EXECUTE] -> (app2:App {systemApp:false})
4 - [event3:WRITE] -> (socket:Resource {type:'socket'})
5 WHERE NOT 'internet' IN app1.permissions
```

Confused Deputy Attacks. We implemented the confused deputy attack described in Section 6.2.2, where a malicious application that does not hold the `INTERNET` permission deputizes the web browser to exfiltrate sensitive contacts data to a remote server. We then verified that the query described previously in Listing 16 indeed correctly identifies this confused deputy attack.

Collusion Attacks. We further implemented two variants of a collusion attack, where two malicious apps coordinate their actions towards a common goal, which in our example is to exfiltrate the SMS database over the Internet. The first malicious app only holds the `READ_SMS` permission, whereas the second app only holds the `INTERNET` permission.

In a simple collusion attack two malicious applications communicate using overt channels, such as Android's Binder IPC mechanism. Listing 17 shows a Cypher query targeting this behavior. We query the behavior graph for non-system applications which do not hold the INTERNET permission (Line 5) and read data from the SMS database (Lines 1 and 2). We search for paths leading to non-system applications, which send data to a remote server via a network socket (Lines 3 and 4). The corresponding subgraph is shown in Figure 17a.



(a) Collusion using Binder IPC (b) Collusion using the file system

Figure 17: Example collusion attacks where two malicious apps coordinate their behavior to exfiltrate the SMS database.

In a more obfuscated collusion attack two applications use covert channels, such as a file shared in the file system, to exchange sensitive data. Note that no direct inter-process communication between both apps occurs in this scenario. Starting from the previous query in Listing 17, we add a file resource node to the query which matches files written to and read from the two colluding applications. Listing 18 shows the resulting query, and Figure 17b depicts a visualization of the discovered subgraph.

Listing 18: Cypher query to detect the collusion attack depicted in Figure 17b

```
MATCH collusion2 = (sms:Resource {type:'sms'})
- [event1:READ] -> (app1:App {systemApp:false})
- [event2:WRITE] -> (file:Resource {type:'file'})
- [event3:READ] -> (app2:App {systemApp:false})
- [event4:WRITE] -> (socket:Resource {type:'socket'})
WHERE NOT 'internet' IN app1.permissions
```

Listing 19: Cypher query to detect the behavior of the "TheTruthSpy" and "LetMeSpy" spyware applications

```
1 MATCH spyware1 = (res:Resource {privacySensitive:true})
2 - [event1:READ] -> (app:App {systemApp:false})
3 - [event2:WRITE] -> (socket:Resource {type:'socket'})
4 WHERE NOT event1.foregroundApp = app.package
```

DroidAuditor can similarly be used to detect signs of collusion attacks via other operating system resources, such as domain or network sockets, ContentProviders or Services. However, it should be noted that DroidAuditor is limited by the granularity of the underlying ASM framework, which is unable to observe app collusion via hardware side channels, such as the CPU cache.

Identifying Spyware Applications. To demonstrate that DroidAuditor is a valid basis for generic application behavior analysis beyond application-layer privilege escalation attacks we installed two popular spyware applications, namely "TheTruthSpy" [259] and "LetMeSpy" [149], on a DroidAuditor device. By analyzing the behavior graph we found that these apps silently access privacy-sensitive resources, such as the CallLogProvider and MMSSMSProvider as well as LocationManagerService, and upload collected privacy-sensitive data to a remote server. We further noticed that these apps only have very limited user interfaces (Activities), which are exclusively used for initial configuration.

To detect such behavior, we first labeled all privacy-sensitive resources in the behavior graph. In Listing 19, we query the graph for non-system applications accessing these resources (Lines 1 and 2) and writing data to a remote server (Line 3). The `WHERE` clause (Line 4) limits our query to apps which silently access privacy-sensitive resources. Figure 18 shows a screenshot of the DroidAuditor client analyzing the corresponding behavior graph.

Detecting Operating System Level Privilege Escalation. Related work has identified malicious applications which attempt to compromise the security architecture of the operating system by exploiting highly-privileged system services or the kernel (see Section 3.3.2). Signs of successful attacks are, for example, the execution of unknown processes with root privileges or applications executing operations they are not authorized for by the permission system. For example, an application connecting to a network server without holding the `INTERNET` permission is an indication of possible operating system level compromise.

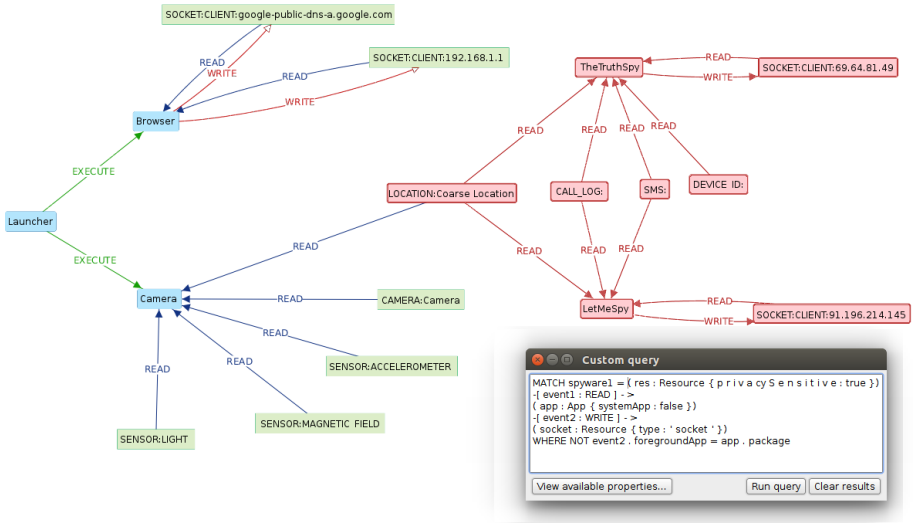


Figure 18: Screenshots of the DroidAuditor client. Here, DroidAuditor is used to analyze the behavior of the "TheTruthSpy" and "LetMeSpy" spyware applications.

The *reliable* detection of such attacks is challenging and out of scope of our adversary model (see Section 3.1), since in general, *all* security solutions which observe application behavior on the operating system level can be defeated by malicious applications operating with the same level of privileges. For example, a strong adversary using kernel or root exploits can adopt rootkit techniques to compromise the trusted computing base of the underlying ASM framework without raising suspicion. Nonetheless, DroidAuditor can potentially be used to identify malicious apps compromising the operating system given a weaker adversary who is not actively trying to hide his traces.

6.2.5 *Conclusion*

In this section, we presented DroidAuditor, a toolkit for long-term application behavior analysis using system-centric access control. Our implementation utilizes the system-centric Android Security Modules access control framework to generate a graph-based model of application behavior. The preliminary evaluation of our proof-of-concept implementation demonstrates that modular access control frameworks are a valid building block for application behavior analysis.

We envision three directions to extend our work. First, an extensive usability study would allow us to better understand how users and security analysts interact with DroidAuditor. Second, the DroidAuditor behavior graph could potentially serve as a building block for a policy enforcement architecture similar to XManDroid [32], where the behavior graph is stored and evaluated on the mobile device. Finally, integrating dynamic taint analysis into DroidAuditor, for example using TaintDroid [66], would enable us to augment the behavior graph with precise information flow data for applications which do not contain native code.

6.3 SECURE DUAL-USE OF SMART MOBILE DEVICES

When analyzing the adoption of smart mobile devices by enterprise customers, a general trend towards a single device for both enterprise and private use can be observed [129]. Two prominent paradigms to implement this trend are *bring your own device (BYOD)* and *corporate-owned, personally-enabled (COPE)* devices. In BYOD, enterprise administrators integrate private employee-owned devices into the enterprise IT architecture. In contrast, the COPE paradigm dictates that devices are owned and distributed by the employer, who grants permission for private use to the employee.

Regardless of the particular paradigm isolation of enterprise and private data and applications on smart mobile devices is of paramount importance to prevent accidental data leakage. Unsurprisingly, both academic research and commercial vendors have proposed a variety of security architectures to isolate enterprise and private applications and data. Android in particular has received much attention due to its popularity [130] and open-source character. The proposed technologies range from isolated enterprise applications [273, 94], kernel-layer compartments [5, 284] to virtualization-based architectures [107, 146, 18, 127]. While the deployment of isolated enterprise applications is straightforward, they rely on the security guarantees of the underlying Android operating system, which is concerning given the amount of well-known attacks (see Chapter 3) and slow deployment of operating system security updates [260]. Furthermore, this approach generally does not support the integration of legacy third-party applications into the enterprise domain. In contrast, architectures based on kernel-layer compartments or virtualization can provide stronger isolation guarantees. However, their adoption on end-user devices is limited since they require the duplication of large parts of the operating system software stack, which is unfavorable given the computation and energy consumption constraints affecting smart mobile devices (see Section 3.4.6).

The BizzTrust solution discussed in this section tackles these challenges by providing security domain isolation for legacy Android applications using mandatory access control. BizzTrust extends our TrustDroid [33] architecture with necessary enterprise features and predates comparable technologies recently introduced by Google (Android for Work [133]) as well as Samsung (KNOX [214]). Today, the award-winning [250, 230] BizzTrust solution is an established commercial product available for a range of Android-based smartphones and tablets.

Contributions. To summarize, our main contributions are as follows:

- **MAC-based Security Domain Isolation.** BizzTrust provides security domain isolation using system-centric mandatory access control on Android.
- **Secure Network Access.** BizzTrust implements secure network access for enterprise apps based on concepts developed by the Trusted Computing community.
- **Commercial Availability.** We discuss interesting aspects of advancing our TrustDroid research prototype towards a commercially viable product.

Remark. Initial versions of BizzTrust were derived from our TrustDroid architecture [33]. Main idea, design, implementation and security evaluation of TrustDroid are due to Sven Bugiel. Bhargava Shastry applied Tomoyo Linux to Android and contributed communication between kernel- and middleware-layer access control mechanisms. The author contributed the design of the application-specific firewall. Alexandra Dmitrienko was involved in discussions about design decisions. Lucas Davi contributed an analysis of related work and participated in general writing tasks. Ahmad-Reza Sadeghi was involved in fruitful discussions and provided feedback, which improved the quality of our publication [119].

The author was further responsible for advancing the TrustDroid prototype towards the award-winning pre-sales version of BizzTrust. He ported the design and implementation to more recent Android versions and additional devices. He further refined TrustDroid's access control architecture and integrated additional policy enforcement points. Finally, the author was responsible for integrating BizzTrust with the Trusted Network Connect (TNC) [263] network admission control technology.

6.3.1 *Scope*

The main goal of BizzTrust is to prevent illegitimate access from private applications installed by the user to enterprise applications and data, and vice versa. To do so, applications are assigned to security domains, such as *enterprise* and *private*. Based on these security domains, BizzTrust enforces a Chinese wall policy [29] to prevent leakage of data across domains. To enforce this policy BizzTrust uses mandatory access control on all layers of the operating system (see Figure 19).

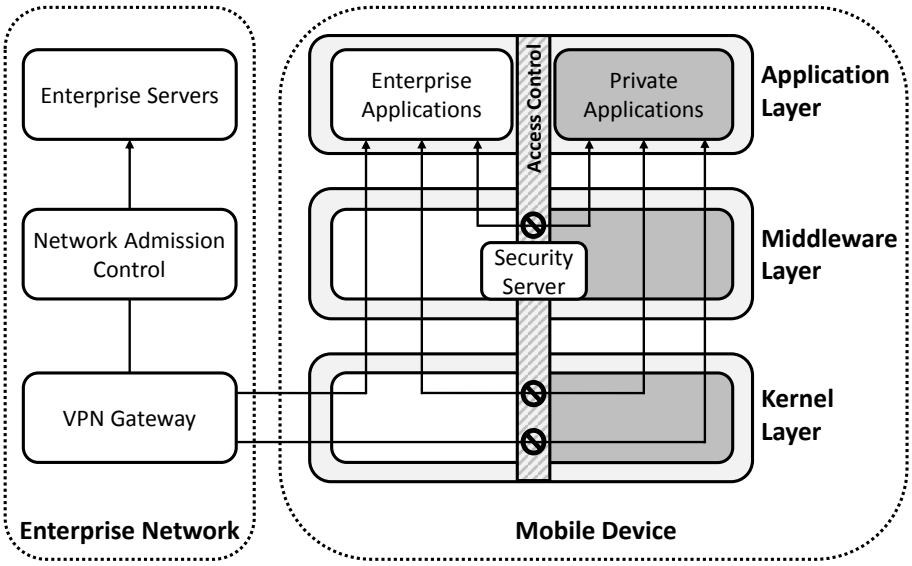


Figure 19: BizzTrust framework design

In accordance with our adversary model presented in Section 3.1 we assume that both users and enterprise administrators can deploy arbitrary Android applications to the device. Further, we assume that the user is non-malicious and does not aim to *intentionally* leak

data. This is reasonable since the user could easily transfer data across domains manually, for example by taking photos of enterprise documents displayed on the screen using an external camera or by manually transcribing sensitive documents. However, the user might accidentally install malicious or privacy-intrusive applications.

Requirements. Given our goal we formulate the following requirements for BizzTrust:

- **Multi-Layer Isolation.** BizzTrust should mediate access to kernel-, middleware- and application-layer resources and communication mechanisms.
- **Legacy Compliance.** BizzTrust should allow both users and enterprise administrators to deploy arbitrary unmodified Android applications to their respective isolated domains.
- **Minimal Performance Impact.** The performance impact of the BizzTrust architecture should be minimal and not disturb the user.

6.3.2 *Design*

In this section, we will discuss the design of our BizzTrust architecture.

High-Level Idea. Our BizzTrust architecture depicted in Figure 19 comprises the following components: First, the middleware-layer SecurityServer serves as the central policy decision point within BizzTrust and manages all installed applications. Second, a set of policy enforcement points within Android's application, middleware and kernel-layer enforce the Chinese wall policy managed by the SecurityServer. Finally, a Network Admission Control solution restricts access to the enterprise network to authenticated devices and provides mobile device and application management.

Security Server. Our BizzTrust SecurityServer associates all installed applications with their corresponding security domains. On the first boot, the SecurityServer assigns the *system* security domain to all preinstalled system applications. Applications installed by the user are assigned to the *private* domain, while authenticated enterprise apps are assigned to the *enterprise* domain. The SecurityServer internally stores the binding between application package name, assigned Linux User ID (UID) and security domain and synchronizes this information with the kernel-, middleware- and application-layer policy enforcement points.

To identify valid enterprise applications during installation our SecurityServer first checks the integrity and authenticity of such applications. To do so, we integrated hooks into Android's PackageManagerService to verify a cryptographic Remote Integrity Metric (RIM) certificate, which we embed into enterprise application packages, using a Mobile Trusted Module (MTM) [262]. RIM certificates are similar to industry-standard X.509 certificates, but their validity can be bound to specific platform security states as defined by the Trusted Computing Group. Accordingly, we assume that the enterprise operates a corresponding public key infrastructure (PKI) to deploy applications.

Finally, our SecurityServer is responsible for signaling the security domain of the current application to the user using visual indicators. Since current Android versions only display

one application at any given time, BizzTrust signals the currently active security domain via Android's StatusBar component. In addition, we use the RGB notification LED available on most Android devices to indicate the security domain in case an application is running in full screen mode where the StatusBar is obscured.

Kernel-Layer Mandatory Access Control. As discussed in Section 2.3.2 Android executes each application inside an isolated sandbox and uses the underlying Linux kernel's discretionary access control mechanisms to isolate applications on the file-system layer. Accordingly, application-private data stored on the internal file system is by default isolated from other applications unless explicitly specified otherwise by the application developer. Finally, Android uses a kernel-layer reference monitor to restrict the use of network and domain sockets to applications holding the `INTERNET` permission.

This approach is however insufficient for enterprise use-cases: First, Android does not enforce fine-grained access control on external storage devices, such as removable Micro SD cards (see Section 4.3.2). Second, app developers might explicitly or accidentally set the file access permissions of their app-private files to world-writable, which could expose enterprise data to private applications. Finally, the granularity of Android's `INTERNET` permission is insufficient to prevent socket-based communication between apps of different security domains.

Kernel-layer mandatory access control (MAC) mechanisms provide the necessary primitives to address these concerns. As described in Section 5.1 the Linux kernel supports a variety of MAC models, such as label [153, 216] and task-oriented [114] access control, via the Linux Security Modules (LSM) interface [283]. These MAC models can at runtime override positive decisions of the default kernel-layer access control model, which in Android's case is discretionary access control (see Section 2.3.2).

Adopting kernel-layer mandatory access control to implement fine-grained security domain isolation on Android is however not straightforward. The reason is that Android consists of three distinguished layers, namely the kernel-, middleware- and application layer. Access control mechanisms deployed on these layers need to operate in concert to prevent data leakage. For example, an application-layer enterprise ContentProvider, which is implemented within userspace, can provide access control enforcement on the ContentProvider interface using Android's default permission model. However, in case the underlying database file is world-readable, private applications could still access sensitive data via the kernel layer.

To address this concern, BizzTrust at runtime synchronizes access control rules between the kernel and the SecurityServer deployed on the middleware layer. To do so, we first integrate a suitable kernel-layer MAC architecture into the default Android Linux kernel. Our SecurityServer propagates the security domain of applications to our kernel-layer access control mechanisms based on the Linux UID assigned to individual applications at installation time. Whenever an application process creates a kernel-layer resource, such as a file or a socket, the resource inherits the security domain of the corresponding application process. The kernel-layer MAC mechanisms at runtime uses both the resource and the application process security domain to decide whether an application process is allowed to access a resource.

Middleware-Layer Mandatory Access Control. Android components primarily rely on Binder IPC communication to interact at runtime (see Section 2.3.1). Accordingly, as a first-step to middleware-layer mandatory access control BizzTrust must prevent applications from communicating via Binder IPC across domains. To do so, we extended Android's default permission-based access control architecture.

Whenever applications establish a communication channel across process boundaries via Binder IPC, Android's middleware-layer ActivityManagerService acts as a reference monitor and enforces the default permission model. The primary mechanism for Binder IPC communication is Intent message passing. Android's ActivityManagerService is responsible for correctly routing these Intent messages. We thus modified ActivityManagerService to enforce our security model on Intent-based communication by dynamically filtering Intent receivers. For example, whenever a private application sends a Broadcast Intent to all installed applications, our modified ActivityManagerService will only forward this Broadcast Intent to system and private applications. Similarly, our modified ActivityManagerService ensures that enterprise applications can only invoke Activities of other system and enterprise applications via Intents. The same concept applies to invocations of Android Services and ContentProviders across domain boundaries.

Since both enterprise and private applications are allowed to communicate with system components and vice versa additional access control mechanisms are necessary. Consider for example that Android's PackageManagerService centrally stores application metadata, such as installed applications and their version numbers. From an enterprise perspective, it is desirable to prevent private applications from learning which applications are deployed within the enterprise domain. Another example is Android's AccountManagerService, which centrally manages credentials for cloud-based services for both enterprise and private applications.

To prevent such information leakage Android components deployed in the system domain are responsible for assigning security domains to resources they manage and to enforce access control on them. Conceptually, this approach is similar to the Userspace Object Managers we adopted in our FlaskDroid architecture (see Section 4.2.3)[2]. More precisely, when processes instantiate middleware-layer resources, the responsible component queries the SecurityServer for the security domain of the corresponding application process and applies the appropriate domain label to the resource. System components are also responsible for labeling incoming data not bound to application processes. For example, SMS/MMS messages are assigned to either the private or enterprise domain based on the security domain associated with the sender's phone number, as we describe in the next paragraph. When application processes later attempt to access these resources, the corresponding system component enforces our Chinese wall policy based on the security domain label of both the resource and application process.

Application-Layer Mandatory Access Control. Our middleware- and kernel-layer mandatory access control extensions prevent private and enterprise applications from establishing communication channels via IPC, the network and the file system. However, several important system components, which are part of Android's application program-

[2]Note that the BizzTrust architecture predates FlaskDroid.

ming interface (API), are implemented not as middleware components, but as system applications deployed on the application layer (see Section 2.3.1). Prominent examples are the contacts and call log ContentProviders. Both private and enterprise applications have legitimate reasons to access these ContentProviders.

To address this issue, we extend such components with additional application layer access control mechanisms, which enforce access control rules based on the security domain of the callee. To do so, our design leverages proxy components located within the system security domain, which can be accessed by both enterprise and private applications. The proxy implements the corresponding Android API and forwards calls selectively to components associated with either the private and enterprise domain depending on the security domain of the application process. In case a system applications requests access to such an API, the proxy queries both the private and enterprise domain components and merges the results.

Secure Network Access. Smart mobile devices used in enterprise use-cases are typically embedded into an enterprise network infrastructure. Following our Chinese wall model, BizzTrust ensures that only authenticated users, devices and enterprise applications can communicate with remote network servers part of the enterprise domain.

Accordingly, BizzTrust establishes a virtual private network (VPN) tunnel to the enterprise network infrastructure, which limits access to the enterprise network to authenticated users. Furthermore, our architecture configures the network packet filter of Android's Linux kernel to prevent private applications from accessing the VPN tunnel.

Additionally, our BizzTrust architecture ensures that only authenticated devices which adhere to a given security policy can establish network connections to the enterprise network. To do so, BizzTrust embraces a concept established by the Trusted Computing community, namely network admission control based on remote attestation. In particular, BizzTrust adopts the Trusted Network Connect (TNC) standard [263]. In TNC, a user's device, denoted *Access Requestor (AR)*, attests its configuration towards an infrastructure-side policy decision point, denoted *Network Access Authority (NAR)* (see Figure 20). If the attestation procedure yields a positive result, the NAR configures a Policy Enforcement Point (e.g., a firewall) to grant access to the protected network. Otherwise, access is denied until the desired state is established.

TNC is a modular standard, where *Integrity Measurement Collectors (IMCs)* deployed on the Access Requestor collect use-case specific integrity measurements. These measurements are evaluated by corresponding infrastructure-side *Integrity Measurement Verifiers (IMVs)*. Matching pairs of IMCs and IMVs are deployed on the TNC client and server and communicate via the *IF-M* protocol [265]. TNC server and client exchange aggregated measurements and access control decisions using the *IF-TNCCS* protocol [267]. Finally, the IF-TNCCS protocol is encapsulated using a use-case specific implementation of the *IF-T* protocol, for example based on Transport Layer Security (TLS) [57, 264] or Extensible Authentication Protocol (EAP) [3, 266].

The generic nature of the TNC specification allows the deployment of a variety of use-case specific IMC-IMV pairs. In our design, we use TNC to attest metadata about the device and specific Android distribution as well as applications deployed within the enterprise environment towards the Network Access Authority.

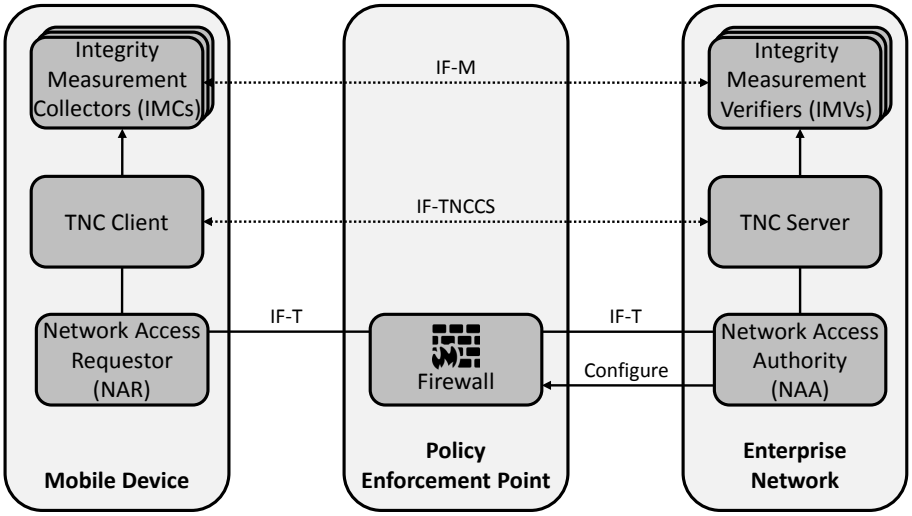


Figure 20: Trusted Network Connect (TNC) integration

6.3.3 *Implementation*

Our initial implementation of BizzTrust was presented to the public in September 2011 at the *it.sa* IT security fair in Nürnberg, Germany. This initial version was based on TrustDroid [33] and targeted Android 2.2.1. The author of this dissertation ported BizzTrust to the more recent Android version 2.3.7 in 2012. Since 2012 Sirrix AG leads the development of BizzTrust and commercially distributes more recent versions targeting Android version 5 and above [229]. BizzTrust today is available for a wide range of smart mobile devices, ranging from Google's *Nexus* line of developer devices to Sony's Xperia series of smartphones and tablets. Due to confidentiality agreements this dissertation however can only provide limited insights into recent developments, and we refer to Sirrix AG for more detailed information.

Security Server. We implemented the BizzTrust SecurityServer within Android's SystemServer, where ActivityManagerService and PackageManagerService reside. This approach prevents unnecessary inter-process communication when middleware-layer components, such as ActivityManagerService and PackageManagerService, invoke the SecurityServer for access control decisions. We further extended Android's application programming interface to enable our middleware- and application-layer access control mechanisms to interface with our SecurityServer via the Context Java class.

We modified Android's PackageManagerService to enable the deployment of enterprise applications. To do so, PackageManagerService at installation time attempts to verify an app-specific RIM certificate. This certificate is generated using the enterprise's public key infrastructure and either deployed as part of the corresponding Android application package or via a mobile device management solution. To this end, we implemented a minimalist mobile trusted module (MTM) within Android's middleware layer. This MTM is preloaded with the necessary enterprise certificate chain to verify corresponding RIM certificates. In case the application's RIM certificate is verified successfully PackageMana-

gerService instructs the SecurityServer to assign the application to the enterprise domain during the installation process. SecurityServer in turn propagates the necessary changes to our kernel-layer mandatory access control framework and configures the Netfilter network packet filter via the `iptables` command line tool.

It should be noted that recent BizzTrust versions adopt an alternative scheme. Current versions of BizzTrust are centrally managed via a mobile device management solution based on an enterprise-side Sirrix Trusted Object Manager [231] appliance. This appliance deploys enterprise applications via an authenticated and encrypted mobile device management channel.

Kernel-Layer Mandatory Access Control. Initial versions of BizzTrust based on Android 2.2 and 2.3 use Tomoyo Linux [114] version 1.8 to instantiate the required kernel-layer mandatory access control mechanisms. Tomoyo provides the necessary primitives to interface with our SecurityServer via the `proc` pseudo-filesystem of the Linux kernel. At boot time, the SecurityServer propagates application UIDs and their security domains via this interface to the kernel-layer. Whenever applications are installed or removed at runtime our SecurityServer synchronizes the corresponding changes with Tomoyo.

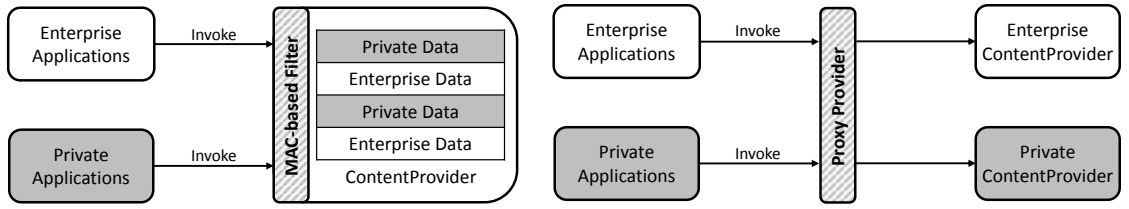
In contrast, recent versions of BizzTrust are based on SELinux [153]/SEAndroid [232] instead of Tomoyo. The rationale behind this decision is that SELinux is the default kernel-layer mandatory access control framework of Android since version 4.3 [101]. While our ASM architecture demonstrates that it is feasible to deploy multiple kernel-layer mandatory access control modules (see Section 5.2.4), rebasing BizzTrust on SEAndroid decreases the required effort for porting BizzTrust to newer Android versions.

Middleware-Layer Mandatory Access Control. Our middleware-layer access control mechanisms primarily concern Android's ActivityManagerService, which we instrumented with additional hooks to query the SecurityServer for access control decisions during Binder IPC transactions. Most notably we introduced hooks which control applications sending (Broadcast) Intents, binding to Services, querying ContentProviders or starting other applications' Activities. In general our access control hooks are invoked after Android's default permission checks have allowed an operation to proceed to avoid unnecessary performance overhead. Additional access control hooks were introduced into Android's PackageManagerService, AccountManagerService as well as telephony and text messaging subsystems.

Application-Layer Mandatory Access Control. We modified important Android system applications to enforce our Chinese wall access control policy separating the private and enterprise domain. Most importantly, we modified several system ContentProviders, such as Android's default ContactsProvider, CallLogProvider and TelephonyProvider.

Regarding ContentProviders it should be noted that our initial TrustDroid implementation [33] stored security domain labels alongside their associated data entries within the corresponding ContentProvider (see Figure 21a). For example, the ContactsProvider uses a SQLite database back end, and we added additional fields indicating the security domain of each entry. However, related work has identified that Android's system ContentProviders are susceptible to SQL Injection attacks [173], which abuse the fact that SQL queries are routinely generated from user- or application-controlled input without proper sanitization. To prevent potential cross-domain data leaks via SQL injection we decided to instead

adopt the proxy-based approach we discussed in Section 6.3.2 (see Figure 21b). In doing so BizzTrust ensures that the effects of potential SQL injection vulnerabilities are contained within the corresponding security domain.



(a) Access Control on ContentProviders using MAC-based filtering

(b) Access Control on ContentProviders using Proxy ContentProviders

Figure 21: Access Control on ContentProviders.

Secure Network Access. Initial versions of BizzTrust embraced the Trusted Network Connect (TNC) architecture described in our design in combination with an IPsec VPN for secure network access. To do so, we first modified Android's VPN subsystem to support split tunneling, which allows private applications to access Internet resources while an enterprise VPN is established. We then deployed a TNC client and corresponding IMC/IMV pairs on the BizzTrust device.

Our Java-based TNC client communicates with an open-source TNC server [112] to attest the state of the BizzTrust device using two IMC/IMV pairs. The first pair authenticates the device based on metadata, such as the unique Android ID, IMEI and the deployed version of BizzTrust. The second IMC/IMV pair collects metadata for applications deployed within the enterprise domain and can additionally be used to push application updates to the device. Once the device corresponds to a given infrastructure-side security policy, access to the internal enterprise network is granted by configuring a Linux-based infrastructure-side firewall using the Netfilter packet filter.

While BizzTrust's initial support of the open Trusted Network Connect standard enabled integration with a variety of TNC-based network admission control solutions more recent versions of BizzTrust opt for an alternative approach. Today, BizzTrust devices are embedded into Trusted Virtual Domains [60], which is a concept for cross-platform security domain isolation and information flow control. Implementations of Trusted Virtual Domain architectures support a variety of device classes, ranging from desktop PCs and servers to smartphones and tablets. Server-side support for central management is provided by the aforementioned Sirrix Trusted Object Manager [231].

6.3.4 *Evaluation*

Performance Overhead. Our initial implementation of the BizzTrust architecture fundamentally inherited the performance and memory overhead of the underlying TrustDroid

architecture [33], and we will briefly summarize these results here[3]. On a HTC Nexus One smartphone the TrustDroid extensions introduced an average overhead of 0.170 ms (standard deviation $\sigma = 1.910$ ms) during Binder IPC compared to the baseline average of 0.184 ms per call when using Android 2.2.1 without our modifications. The high standard deviation is caused by high system-load due to heavy multi-threading on the single-core processor of the Nexus One smartphone. The signature verification during the application installation process caused an average 869.750 ms of overhead (standard deviation $\sigma = 645.313$ ms).

Security Consideration. BizzTrust efficiently isolates applications using mandatory access control. On the kernel-layer, the Tomoyo and SELinux frameworks effectively prevent applications from direct communication via Linux IPC mechanisms, network sockets and the file system. Our middleware layer enhancements prevent applications from using Binder IPC to directly communicate. While BizzTrust generally does not prevent application-layer privilege escalation attacks, such as confused deputy and collusion attacks, it contains their effects to the corresponding security domain.

Our architecture does not consider communication across domains via covert channels. Given our security model and goals (see Section 6.3.1) excluding such attacks is reasonable, since to establish a covert communication channel across domains the adversary would have to deploy applications into *both* the enterprise and private domain. While the adversary might use social engineering or other attack vectors (see Section 3.2) to deploy private applications, our changes to PackageManagerService prevent him from deploying applications to the enterprise domain. Further, it is reasonable to assume that enterprise applications would not deliberately provide covert channels for cross-domain communication since they are inherently trusted by the enterprise. Finally, our network admission control architecture and per-application packet filter prevent unauthorized users, applications and devices from accessing the enterprise IT infrastructure.

Finally, we note that while BizzTrust is an effective and efficient mechanism to isolate enterprise and private applications on Android based devices there are alternative approaches which provide even higher isolation guarantees. The reason is that compared to solutions based on virtualization on minimized operating system kernels and hypervisors [107, 146] or kernel-layer compartments [5, 284] the trusted computing base (TCB) of BizzTrust is larger (see Section 3.4.6). BizzTrust's TCB includes Android's kernel, middleware components as well as system applications which provide services to both the enterprise and private domain. Accordingly, BizzTrust exposes a considerable attack surface to applications deployed within the private security domain. However, the proposed alternative approaches based on virtualization and kernel-layer compartments inherently duplicate a large part of the software stack, which leads to an increased consumption of memory and CPU time and ultimately effects battery lifetime. While SoCs for smart mobile devices today even feature efficient virtualization extensions, adoption of these mechanisms in practice is still very limited. In contrast, solutions based on system-centric access control mechanisms, ranging from Android for Work [133] to Samsung KNOX [214]

[3]Note that due to confidentiality agreements we cannot publish corresponding results for more recent versions of BizzTrust

and BizzTrust are deployed on millions of devices and partially have been certified for handling classified confidential resources in highly sensitive environments [117].

6.3.5 *Conclusion*

In this section, we discussed the BizzTrust security domain isolation architecture. BizzTrust adopts system-centric mandatory access control mechanisms on all layers of the Android operating system to effectively and efficiently isolate private and enterprise applications. While the underlying Tomoyo-based multi-layer access control framework was first presented in our TrustDroid architecture [33] modern versions of BizzTrust adopt and augment SELinux type enforcement with additional application- and middleware-layer access control mechanisms.

BizzTrust serves as an example which demonstrates the challenging process of commercializing system-centric access control extensions. Indeed, integrating invasive changes into Android's default security architecture exacerbates the process of adapting to new releases of the operating system. Nonetheless, the success of particular academic security extensions, ranging from the SELinux/SEAndroid [153, 232] mandatory access control architecture to the award-winning BizzTrust security domain isolation solution [250, 230] demonstrate that despite practical challenges in mobile platform security it is feasible to advance research prototypes to commercial products.

6.4 ACCESS CONTROL IN ADVANCED IOT SCENARIOS

In the previous sections, we have shown that system-centric access control architectures, such as FlaskDroid, Android Security Modules and BizzTrust, are suitable solutions for a variety of interesting use-cases. However, while system-centric solutions have favorable security properties compared to application-layer approaches, their initial deployment requires a modification of the Android operating system. Thus, these solutions should ideally be deployed by either Google or the mobile device manufacturer. Unfortunately, only few academic projects have been embraced by industry. Important examples are the SEAndroid mandatory access control architecture [232], which today is part of mainline Android, and TrustDroid [33], which laid the foundations for the commercial BizzTrust solution (see Section 6.3).

Access control architectures based on application-layer deputies [136] are an interesting alternative for situations where deployment of system-centric solutions is not feasible. While they provide lesser security guarantees their deployment is straightforward, since they do not introduce changes to the operating system (see Section 3.4.7). Further, due to their design these approaches are ideal candidates for scenarios targeting different hardware platforms and legacy operating systems.

Consider for example the Internet of Things (IoT), where a variety of different classes of devices in different form factors, ranging from personal information and entertainment devices, such as smartphones, tablets, smart TVs and wearables, to automotive head units for smart cars are being equipped with increasing computing, storage and wireless communication capabilities. The Internet of Things promises to intelligently interconnect these devices, which allows applications to adapt to available resources in the environment and to share their capabilities to improve the user-experience and maintainability significantly: For example, imagine placing a video call from a smartphone using a nearby Android TV [96] as a display; a smartphone navigation app using the more precise GPS sensors and larger display of the head unit available in a modern vehicle; letting a navigation app direct an autonomous vehicle, or resource-constrained devices outsourcing computationally expensive tasks (e.g., object recognition) to other more powerful devices.

However, today the ability for such intelligent and adaptive device collaboration falls short. Current network discovery and media sharing protocols, like UPnP [258], DLNA [59], Apple Airplay [128] or Samsung AllShare [213], limit themselves to a set of predefined services. More sophisticated use-cases for advanced device collaboration, be it in the area of smart vehicles, smart buildings or personal entertainment, require custom software components that have to be installed, managed and configured individually on each device. This is tedious, time consuming, and poses security and privacy risks. More flexible solutions for collaboration among devices based on migrating code from one device to another [192, 92, 49, 143] do not adequately address security and privacy concerns.

In this section, we present Xapp, a context-aware service mobility framework, which enables intelligent and secure resource sharing between advanced IoT devices. Xapp differs from prior work on distributed cross-device functionality in two major aspects. First, it provides fine-grained access control on sensitive resources using a lightweight token-based authentication and authorization system. Our access control architecture uses application-

layer deputies and does not require changes to the underlying operating system. Second, it allows users to keep sensitive assets on their trusted devices. By adopting standard technologies where possible, Xapp supports multiple COTS operating systems and can be deployed either as a system-centric platform component or installed as an app without changes to the underlying operating system.

Contribution. To summarize, our main contributions are as follows:

- **Flexible Distributed Services.** We design the Xapp architecture, which enables users to securely run an Android app across multiple devices without having to install it on each of them individually.
- **Intuitive Authentication and Authorization.** Xapp provides an authentication and authorization protocol suite where trust is bootstrapped using Near Field Communication (NFC) for the sake of usability.
- **Proof-of-Concept Implementation.** We implement a prototype of Xapp on the service-based R-OSGi [257, 205] software stack, an emerging industry standard which we extended with mechanisms for fine-grained access control and secure communication.

Remark. The results presented in this section were achieved in collaboration with Christoph Busold, Jon Rios, N. Asokan and Ahmad-Reza Sadeghi. Main idea and high-level design are due to the author. Jon Rios provided the initial implementation with additional contributions by Christoph Busold, who focused in particular on cryptographic protocols. Evaluation was performed by both Jon Rios and Christoph Busold. The author was tasked with writing our publication [120] in collaboration with Christoph Busold. Both Ahmad-Reza Sadeghi and N. Asokan were involved in fruitful discussions and provided feedback which improved the quality of our publication.

6.4.1 *Scope*

The main goal of Xapp is the controlled distribution of software modules to multiple devices in advanced IoT scenarios. These software modules share resources across devices and are subject to strict access control. To describe Xapp we first present our system model (see Figure 22), which involves the following entities:

- **A host** $\mathcal{H}$ provides resources R to other devices (e.g., a smart TV sharing its screen, camera and microphone).
- **A manager** $\mathcal{M}$ issues a cryptographic access token T to grant access to resources R on a host $\mathcal{H}$ to other devices (e.g., the smartphone of the smart TV's owner). Accordingly, we assume that every host $\mathcal{H}$ trusts its manager $\mathcal{M}$ and vice versa.
- **A client** $\mathcal{C}$ initiates communication with $\mathcal{H}$ and distributes parts of its application to $\mathcal{H}$ in order to use resources R according to an access control policy specified within the token T.

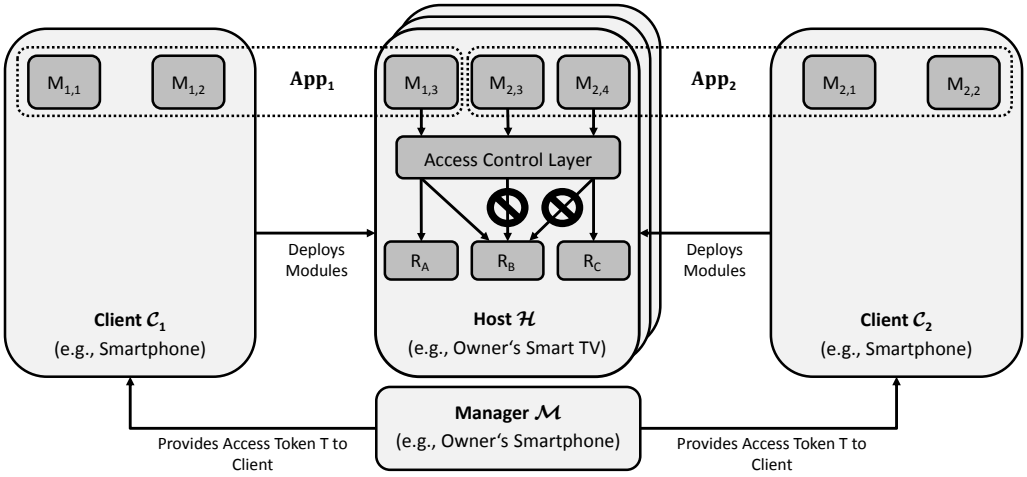


Figure 22: Xapp system model: Entities and interaction

In our model entities are devices in a network, which are identified by their IP addresses and can be discovered using a suitable service discovery protocol, such as Simple Service Discovery Protocol (SSDP) [258]. An application is partitioned into a set of software modules M, which represent different tasks implemented by this particular application and depend on a set of available resources R. This module-based approach is in line with component-based programming models used, for instance, by the Android operating system (see Section 6.4.3).

Refined Adversary Model. Since our Xapp architecture focuses on access control in distributed environments of multiple communicating software modules we need to refine our generic adversary model presented in Section 3.1 to include attacks on network communication. In particular, Xapp considers the following adversaries:

External attackers $\mathcal{A}_{ext}$ are classical Dolev-Yao adversaries [61]: They do not have access to any of the devices or application modules M, but have full control over the network and thus can eavesdrop on, manipulate, inject and replay messages. Such attacks could be used, for example, to inject malicious code into an application module, which is transmitted to another device.

Each client $\mathcal{C}$, host $\mathcal{H}$ or application module can potentially be an *internal attacker* $\mathcal{A}_{int}$, resulting in two possible scenarios. First, a malicious client $\mathcal{C}$ can send a malicious module to a host $\mathcal{H}$ in order to gain unauthorized access to resources R and sensitive information, or even infect the platform or other modules M on $\mathcal{H}$. Xapp should mitigate attacks from the malicious module on $\mathcal{H}$ or any other application modules M running on it. Second, a malicious host $\mathcal{H}$, hosting application modules M, may attempt to compromise the client application, for example by tampering with modules M running on $\mathcal{H}$. Xapp should support the developer in protecting his application and sensitive user data against such attacks by storing and processing sensitive data only on the user's trusted device (e.g., his smartphone acting as client $\mathcal{C}$).

Requirements. Given our adversary model and goal, we formulate the following requirements for Xapp:

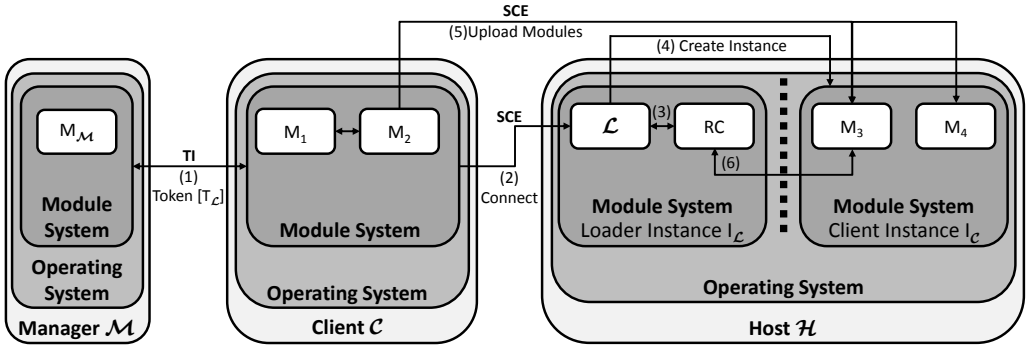


Figure 23: Xapp framework design

- **Protection of User-private Data.** A user's sensitive data, applications and modules M located on the user's own device (client $\mathcal{C}$), are protected from the internal adversary $\mathcal{A}_{int}$ on a host $\mathcal{H}$.
- **Host Protection.** $\mathcal{A}_{int}$ can neither compromise other sensitive applications nor modules M and their data on a connected host $\mathcal{H}$.
- **Protected Communication Channels.** $\mathcal{A}_{ext}$ cannot gain access to any resource R by eavesdropping on or manipulating the network channel.
- **Low Performance Overhead.** Xapp does not introduce excessive performance overhead, meaning minor user interaction and the capability to automatically move modules M to a host $\mathcal{H}$.
- **Platform Independence.** Application modules M should run independently of the underlying hardware and operating system. This requires compatibility with common operating systems. Ideally Xapp should run as a third party application.

6.4.2 *Design*

In this section, we present the design of our cross-device application framework Xapp. It comprises a security architecture for sandboxing modules of different applications and stakeholders, a generic resource control concept and a token-based authentication and authorization system.

Architecture Overview and High-Level Idea. Our generic architecture is shown in Figure 23. On every host $\mathcal{H}$ a component called Loader $\mathcal{L}$ manages modules M running on $\mathcal{H}$ and their privileges to access resources R on $\mathcal{H}$. This Loader $\mathcal{L}$ is initially installed and configured on each host, either by the device owner or by the device vendor. The owner takes ownership of $\mathcal{L}$ by establishing a shared symmetric key $K_{\mathcal{M}}$ between the manager $\mathcal{M}$ (e.g., his smartphone) and $\mathcal{L}$. For our implementation we use a key agreement protocol over NFC due to the required physical proximity [116]. This approach is similar to the *resurrecting duckling model* [240], where physical contact is used to create a binding between two entities.

Xapp enables the developer to encapsulate the functionality of an application on a client $\mathcal{C}$ into a set of modules M, which potentially use resources on a remote host $\mathcal{H}$. We implemented an adaptation of the *extended duckling model* [239] to control which clients may upload modules to $\mathcal{H}$, and which resources may be used by a client $\mathcal{C}$. When a client $\mathcal{C}$ wants to use resources R of $\mathcal{H}$, it first requests an access token $[T_{\mathcal{L}}]$ from $\mathcal{H}$'s manager $\mathcal{M}$ (Step 1) using the Token Issuing protocol (TI). $\mathcal{C}$ authenticates to $\mathcal{L}$ using the Secure Channel Establishment protocol (SCE) with this access token (Step 2), which is forwarded to the Resource Controller RC (Step 3). Next, $\mathcal{L}$ creates a restricted execution environment $I_{\mathcal{C}}$ (Step 4) for modules M uploaded by $\mathcal{C}$ (Step 5). Modules which trust each other (e.g., modules belonging to the same application) may share an instance. Modules run inside their instance $I_{\mathcal{C}}$, which provides life-cycle management. $\mathcal{L}$ is executed inside a privileged instance $I_{\mathcal{L}}$ with access to all resources.

In Xapp instances are created on demand and removed when they are no longer needed, e.g., because their modules are removed. To protect $\mathcal{H}$ from malicious modules of the internal adversary $\mathcal{A}_{int}$, instances $I_{\mathcal{C}}$ follow the principle of least privilege, meaning that direct access to resources is limited to what is basically required by their modules. When a module aims to access shared resources on $\mathcal{H}$, it queries RC located inside $I_{\mathcal{L}}$ (Step 6). RC mediates access to resources R based on a policy $P_{\mathcal{C},\mathcal{H}}$ included in the token $[T_{\mathcal{L}}]$, as described in the following section.

Resource Control Concept. When the manager $\mathcal{M}$ creates an authentication token $[T_{\mathcal{L}}]$ for a client $\mathcal{C}$, it can bind an access control policy $P_{\mathcal{C},\mathcal{H}}$ to this token. Policies are forwarded by the Loader $\mathcal{L}$ to the Resource Controller RC, which is responsible for their enforcement on $\mathcal{H}$. A policy consists of a set of individual privileges. Each privilege $\mathsf{Privilege}(\mathsf{R},\mathcal{C},\mathcal{H},\mathsf{S}) = \mathit{Yes} \,|\, \mathit{No} \,|\, \mathit{Ask}$ describes whether or not the instance $I_{\mathcal{C}}$ may access a particular resource R on $\mathcal{H}$, optionally limited to a given state S (e.g., time of day). The *Ask* value specifies that $\mathcal{H}$ should consult $\mathcal{M}$ at runtime when $I_{\mathcal{C}}$ tries to access this resource. Policies can further contain optional lifecycle constraints to address possible resource starvation attacks by malicious modules. For example, $\mathcal{M}$ can define that a shared resource is only accessible for a specified amount of time, or that $I_{\mathcal{C}}$ should be removed after a certain time span.

Consider the video call use case, where $\mathcal{M}$ creates a policy restricting the access of $\mathcal{C}$'s instance $I_{\mathcal{C}}$ to the camera, microphone and screen of the smart TV $\mathcal{H}$, thereby protecting the privacy of the smart TV's owner. Modules installed by $\mathcal{C}$ are denied access to other sensitive resources, such as photos accessible by the TV. Finally, $\mathcal{M}$ uses a state-aware policy to allow $I_{\mathcal{C}}$ to access the camera and microphone only when the video call module is running in the foreground on $\mathcal{H}$, and to automatically remove $I_{\mathcal{C}}$ after one hour.

Authentication and Authorization Protocols. Our design includes a flexible and secure protocol suite providing for authentication of clients, authorization for resource access and security on the communication links. This protocol suite is based on standard cryptographic primitives and provides offline verification, i.e., the access control token is verifiable by $\mathcal{H}$ if its manager $\mathcal{M}$ is not available. Offline verification can be achieved by token-based protocols such as Kerberos [172]. However, Kerberos requires a database with known clients, which is managed outside the protocol. Therefore we designed a custom

token protocol, which can handle both ad-hoc as well as long-term clients and at the same time reduces the complexity of Kerberos. A detailed description of our protocols is out of scope of this dissertation. Hence, we will only provide an overview and refer to our publication [36] for a more detailed discussion.

Our protocol consists of two parts (see Figure 23). During the *Token Issuing Protocol* (TI) $\mathcal{M}$ issues a Token $[T_{\mathcal{L}}]$ to $\mathcal{C}$. $[T_{\mathcal{L}}]$ is bound to a key $K_{\mathcal{C}}$, which is computed through a Diffie-Hellman key agreement scheme DH between $\mathcal{M}$ and $\mathcal{C}$. $\mathcal{C}$ uses $[T_{\mathcal{L}}]$ to authenticate itself to $\mathcal{L}$ and to establish a secure channel using the *Secure Channel Establishment Protocol* (SCE). It proceeds to request a new execution instance $I_{\mathcal{C}}$. Finally, $\mathcal{C}$ uses the SCE protocol to connect to $I_{\mathcal{C}}$ by creating a new token $[T_{I_{\mathcal{C}}}]$ encrypted by K_C and with a randomly chosen key K_I inside. The only setup requirement is a shared symmetric secret key between $\mathcal{M}$ and $\mathcal{H}$, denoted $K_{\mathcal{M}}$, which is used to authenticate and encrypt tokens with the help of an authenticated encryption scheme AE. This secret key $K_{\mathcal{M}}$ has been established during the initial pairing between $\mathcal{M}$ and $\mathcal{H}$.

Interactive Privilege Evaluation. As described previously resources protected by an *Ask* privilege require runtime consultation of the manager $\mathcal{M}$. For that purpose, the relevant host $\mathcal{H}$ sends the identity of $\mathcal{C}$ and the identifier of the resource R together with a nonce $\mathcal{N}$ to $\mathcal{M}$. To secure the authenticity of $\mathcal{M}$'s responses, $\mathcal{M}$ computes a message authentication code (MAC) over the decision value and the original request including the nonce $\mathcal{N}$ using the shared secret key $K_{\mathcal{M}}$. If $\mathcal{H}$ fails to verify this MAC or does not receive a response at all within a certain time frame, it defaults to deny the request.

Access Revocation. Since our solution focuses on time-limited deployment of cross-device applications via lifecycle constrains we do not consider revocation in our current implementation. However, token revocation could be added to Xapp by means of revocation lists. The integrity and authenticity of revocation lists can be assured using MACs based on a key derived from $K_{\mathcal{M}}$. Alternatively, we could adopt a token status protocol comparable to OCSP [215], but since Xapp is designed for offline token validation we deem revocation lists to be better suited.

6.4.3 *Implementation*

Our implementation of the Xapp design is based on the Open Service Gateway Initiative (OSGi) specification [257], which is a widely-deployed platform-independent industry standard for Java software modularization. We run our implementation on Android, which serves as an example of a modern operating system for advanced IoT devices. In this section, we highlight the technical challenges we had to tackle and describe several security extensions we developed for the R-OSGi remote procedure call (RPC) layer [205]. Figure 24 shows the instantiated components.

Platform Considerations. We implemented the module system based on Apache Felix [252], which is an open-source implementation of the OSGi specification. OSGi allows us to easily integrate existing solutions that can extend our framework with further desired functionality (such as service discovery protocols [71, 258, 253]), which is orthogonal to our work. OSGi divides applications into modules, called *bundles*. A bundle is a collection

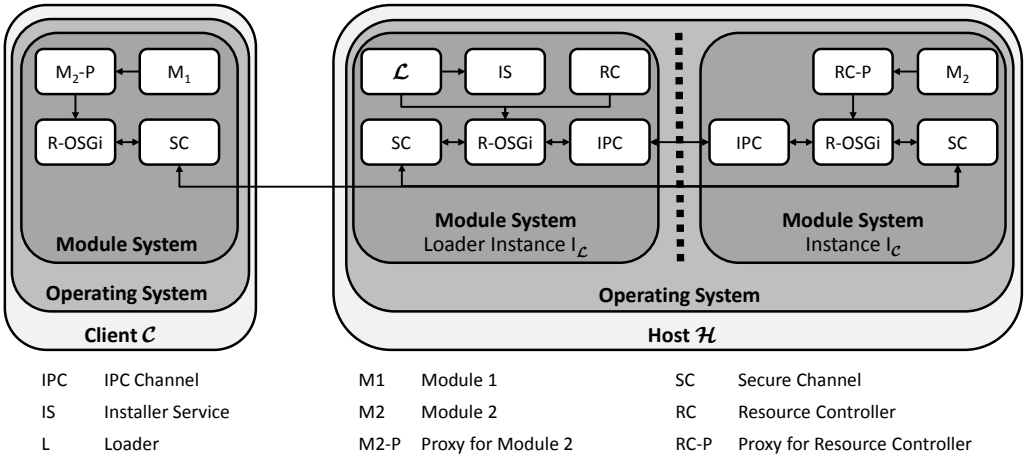


Figure 24: Xapp framework implementation

of self-contained Java packages, arbitrary data and a manifest file. This manifest contains metadata about the bundle along with its platform requirements, provided services and dependencies on other bundles. At runtime, bundles interact via OSGi services, which can be published to and consumed by other bundles.

Further, the adoption of OSGi enables us to seamlessly connect modules on different devices using the remote service layer of Remote OSGi (R-OSGi) [205]. R-OSGi extends the concept of services in OSGi to remote services, which can be published to and accessed from other framework instances, possibly running on different machines. At runtime, R-OSGi can connect to other hosts running R-OSGi and query them for available remote services. Finally, the platform independence of OSGi allows us to instantiate our framework on a wide range of operating systems for advanced IoT devices with different capabilities, ranging from mobile devices and automotive head units to desktop PCs and virtual machines in cloud environments. Individual security aspects of the target operating system (most importantly application isolation and access control) must be considered when adopting our framework. For example, Android relies on process-level permissions and per-app sandboxes. On PCs, IBM's Java JVM 8 provides a multi-tenancy environment [140], which efficiently isolates Java applications executed in one Java VM and uses the Java Security Manager [182] for access control. Another approach particularly interesting in the context of cloud computing environments is Maxine Virtual Edition [181], which instantiates isolated Java runtime environments on top of the Xen hypervisor.

For our prototype implementation we selected Android as the target platform not only because it is the most popular platform for smartphones and tablets [130], but also because it is deployed in other IoT market segments, e.g., automotive (Android Auto [132]) and home entertainment (Android TV [96]). While platform documentation on Android Auto is currently limited, Android TV is a standard Android distribution optimized for large screens and thus allows the deployment of Xapp without further modifications.

Loader. The Loader $\mathcal{L}$ (see Figure 24) is $\mathcal{H}$'s interface to an external client $\mathcal{C}$ and exposes its functionality to remote and local application modules via R-OSGi remote services. This allows clients to create, remove, start and stop their instances on a host and to deploy

application modules, as explained in the following. The Loader is implemented as a set of OSGi bundles – most importantly the *Installer Service* and *Resource Controller* bundles – which run inside a privileged instance $I_{\mathcal{L}}$.

Installer Service. The *Installer Service* IS is used by $\mathcal{L}$ to create and remove client instances on $\mathcal{H}$. While the implementation of the Installer Service IS is platform-specific it communicates via a standard OSGi service interface. On Android, instances are implemented as Android applications and isolated by Android's default sandboxing mechanism (see Section 2.3.2). Our Installer Service IS for Android bootstraps new instances from a template application in the form of an Android Application Package (APK). This APK includes the Apache Felix OSGi framework and required bundles to communicate with $\mathcal{C}$ (e.g., R-OSGi). Since Android apps are identified by a unique package name the Installer Service IS rewrites the application package file with a new name and configures parameters specific to the new instance $I_{\mathcal{C}}$, such as the listening port of R-OSGi.

When app modules are deployed by a client $\mathcal{C}$, the installation of the client-specific instance $I_{\mathcal{C}}$ is ideally performed silently without user interaction once $\mathcal{H}$ has validated $\mathcal{C}$'s access token. Due to Android-specific limitations this is only possible if $\mathcal{L}$ is an Android system app: For security reasons third-party apps cannot install or remove other apps silently on stock Android. More precisely, third-party apps cannot obtain the *SystemOrSignature*-level `INSTALL_PACKAGES` permission (see Section 2.3.2). Thus, if the Loader is installed on a stock Android device, Xapp requires minimal user interaction, since the user has to approve the installation of $I_{\mathcal{C}}$ by clicking a button on $\mathcal{H}$.

Resource Controller. The *Resource Controller* RC exposes resources of the host $\mathcal{H}$ to a client's instance $I_{\mathcal{C}}$ using a R-OSGi service. RC is executed inside the privileged Loader instance $I_{\mathcal{L}}$, which holds all permissions required to access security- or privacy-sensitive resources R (e.g., contacts or camera) exposed to instances I. Access to these resources is mediated according to the instance-specific access control policy contained within the token $[T_{\mathcal{L}}]$. The implementation of the Resource Controller is platform-specific, while its interface is the same on different operating systems.

In general, there are two possible approaches to implement access control on resources: Either the Resource Controller RC uses the existing platform-specific access control mechanisms, or RC implements the required access control hooks itself. Both approaches have advantages and disadvantages:

In the former case, RC maps privileges to operating system specific access control mechanisms. As discussed in Section 2.3.2 Android uses *permissions* for access control on APIs as well as *discretionary* and *mandatory* access control for kernel-level resources. More advanced architectures, such as FlaskDroid or Android Security Modules (see Chapters 4 and 5) provide interfaces to influence system-level access control decisions at runtime and could potentially be integrated with Xapp. However, such an integration would limit our solution to modified versions of the Android operating system and thus violate our interoperability requirements.

In the latter case, RC implements access control on resources itself by acting as an application-layer deputy for the unprivileged instance $I_{\mathcal{C}}$. We selected this approach, since it allows Xapp to consider fine-grained and state-aware access control policies without

introducing changes to the underlying operating system. In Section 6.4.4 we will discuss security implications of this design decision.

Our Extensions to Remote OSGi. Our implementation provides several security extensions to the Remote OSGi (R-OSGi) framework, as described in the following.

The R-OSGi middleware offers by default only TCP communication and provides no protection against an external attacker $\mathcal{A}_{ext}$. To enable secure communication between the Loader instance $I_{\mathcal{L}}$, the client $\mathcal{C}$ and his remote instance $I_{\mathcal{C}}$, we extended R-OSGi with a secure network channel (SC), which protects both confidentiality and integrity of messages using authenticated encryption with a symmetric key (see SCE protocol Section 6.4.2). To provide efficient communication between different OSGi frameworks in separate sandboxed instances on the same host $\mathcal{H}$ (e.g., $I_{\mathcal{C}}$ and $I_{\mathcal{L}}$), we further implemented an IPC communication channel using domain sockets.

Further, since the original design of R-OSGi does not differentiate between different communication channels, a service can only decide whether it will be published via R-OSGi or not. If so, it is always registered on *all* available communication channels. This is insufficient, if one wants to expose a service only via the IPC channel to local instances running on the same host $\mathcal{H}$. To address this limitation we extended the original R-OSGi implementation with a filter on communication channels, so that services can choose the channels they are available on.

Another challenge is that services do not know over which channel and from which endpoint they were called (i.e., remotely or locally), because this connection between the function call and the originating channel endpoint is hidden by the abstraction of R-OSGi. In our model, this information is crucial in order to decide whether access to a service should be granted or not, depending on the identity of the caller (i.e., modules M of a client $\mathcal{C}$ executed in instance $I_{\mathcal{C}}$). We thus implemented a function to retrieve the identity of the current caller from R-OSGi. For the IPC channel we get the Linux UID of the connected process and in case of the secure network channel we extract the identity from the token that was used during the channel establishment protocol.

Xapp Development Model. To support Xapp, apps need to adhere to the (R-)OSGi programming model, since Xapp is not limited to one specific operating system. Specifically, developers need to integrate an OSGi runtime environment into their applications, such as the open-source implementation *Apache Felix* [252], on top of which the Xapp components (mainly R-OSGi and the Loader) as well as application-specific bundles are executed. Consequently, application modules which should migrate between hosts need to be implemented as OSGi bundles. These bundles at runtime communicate with other bundles on the local or remote host via OSGi services. OSGi services are comparable to Android services in that they adhere to the same RPC communication paradigm. To ease the work of developers who want to adopt our solution, Xapp provides a set of service definitions for common resources, such as *contacts information*, *camera* and *microphone*.

6.4.4 *Evaluation*

In this section, we evaluate Xapp in terms of performance, usability, interoperability and security.

Step	Average Time (in ms)	Standard Deviation σ (in ms)
Signature Updating	230.66	15.38
Instance Installation	995.70	23.54
Instance Startup	672.80	24.81
Packaging	728.80	35.51
Total	2,627.96	99.24

Table 10: Performance evaluation (framework)

Performance. To evaluate the performance of different Xapp components we deployed our proof-of-concept implementation on two Samsung Galaxy S3 I9300 smartphones running Android 4.0.4 (client and manager) and a Nexus 10 Tablet running Android 4.2.1 (host). All devices communicate using an 802.11bgn WiFi network. We use the industry standard algorithms AES-256 in EAX mode as authenticated encryption scheme AE and ECDH-256 as key exchange protocol DH.

For the Android-based implementation of the token issuing protocol TI we measured the elapsed time between starting the communication with the manager $\mathcal{M}$ and receiving the token $[T_{\mathcal{L}}]$. Overall the protocol takes 308.28 ($\sigma = 27.73$) ms on average over 20 runs.

For our framework we first performed micro benchmarks to measure the time required for creating a basic instance containing no bundles. This includes all steps starting from verifying the access token $[T_{\mathcal{L}}]$, creating the application package, installing and finally starting the instance I. The results are presented in Table 10 and show the average and the standard deviation σ of the time required to perform all steps over 20 runs. These numbers are reasonable considering that our implementation has not been optimized for performance yet, and we refer to our case study below for further discussion of these results. Further, we verified that the OSGi framework incurs no noticeable performance overhead at runtime using the Java Linpack benchmark [64] both in a regular Android app and in a Xapp module. The average performance over 20 runs is 143.15 ($\sigma = 0.13$) MFlops and 137.41 ($\sigma = 0.33$) MFlops respectively, which shows a small difference of 4.18%.

We also performed micro benchmarks to measure the performance impact introduced by our access control architecture. In our Android-based implementation we query the contacts database to retrieve a single contact both in a regular Android app and in a Xapp module. The process takes on average 17.47 ($\sigma = 4.41$) and 65.50 ($\sigma = 3.86$) milliseconds respectively over 1000 runs. The high standard deviation is caused by varying system load. The difference of around 48 milliseconds introduced by the redirection of calls via the Resource Controller RC and the access control enforcement is partially caused by marshalling the data over the domain socket. This overhead can potentially be reduced by mapping the memory into the process, for example using Binder IPC, instead of copying the individual contacts data fields.

Interoperability and Portability. Our design enables the isolation of modules deployed on any operating system and hardware platform which provide adequate sandboxing and privilege separation capabilities. Since our framework operates on the application layer, it requires no changes to the operating system, as demonstrated by our implementation on Android. When an Android device vendor deploys Xapp, it is even possible to install new instances without user interaction by installing the Loader as a *system* app.

It should be noted that while we instantiated our framework on Android, our architecture only requires a standard-compliant Java Runtime Environment with an OSGi framework and a platform-dependent isolation and privilege separation mechanism (see Section 6.4.3). Java is used on a variety of operating systems and platforms, from smart mobile devices to mainframes, and open-source implementations of the Java Virtual Machine and OSGi standard are available.

Usability. Pairing of devices via NFC has been adopted for a wide range of consumer devices, such as printers and Bluetooth speakers. Our performance measurements indicate that the time required to deploy app modules on one or more hosts (see Tables 10 and 11) is reasonable considering the alternative, which is to manually search, install and later uninstall an app on each host. While the definition of access control rules in the manager app is straightforward, one possible limitation is that with a growing number of rules a user might be tempted to always allow any requests for access to privileges by a client [80]. However, since the functionality of app modules is limited and tailored to specific use cases, they only need access to a limited set of resources, which accordingly limits the number of privileges a manager has to consider.

Proof of Concept: Video Call Application. To demonstrate the advantages and feasibility of our solution we implemented the video calling use case, where Alice uses her smartphone (client $\mathcal{C}$) and Hector's Smart TV (host $\mathcal{H}$) to place a video phone call to Bob. This use case highlights an advantage of app partitioning over just connecting the TV to the phone: The video stream does not have to be routed through Alice's smartphone, but can be processed and sent to the TV directly by Bob's smartphone. Furthermore, Xapp protects Alice's privacy in case the TV is untrusted, since she does not have to enter her login credentials on the potentially malicious smart TV. Instead, she can keep sensitive data (e.g., login credentials or contacts information) on her trusted client device.

Step	**Average Time (in ms)**	**Standard Deviation σ (in ms)**
Signature Updating	330.86	12.86
Instance Installation	1,122.16	32.47
Instance Startup	2,228.47	48.49
Bundle Transmission	1,837.77	49.79
Packaging	1,271.20	81.16
Total	6,790.46	106.43

Table 11: Performance evaluation (case study)

Table 11 presents the performance evaluation results for creating an instance within this use case. In contrast to the previously discussed basic instance, these numbers contain a

transmission phase, where modules with an overall size of 34.2 kB are sent to the host and added to the installation package, which increases the startup time. In total, our unoptimized case study implementation requires about 6.79 seconds to deploy the relevant app modules, which is comparable to downloading and installing apps via an app market. Note that a client $\mathcal{C}$ can deploy modules on multiple hosts in parallel and in contrast to classic applications our cross-device apps do not require any further lifecycle management, such as updates and configuration, on the involved devices.

Security Considerations. We now discuss how the security mechanisms implemented in Xapp allow us to achieve the security goals defined in Section 6.4.1.

An *external adversary* $\mathcal{A}_{ext}$ needs valid tokens to gain access to either the loader $\mathcal{L}$ on host $\mathcal{H}$ or an instance $I_{\mathcal{C}}$ deployed by a client $\mathcal{C}$. The initial pairing between $\mathcal{H}$ and the manager $\mathcal{M}$, during which a shared symmetric key $K_{\mathcal{M}}$ is established, is performed using confidential and authenticated communication. In our implementation we establish this key over NFC, which is resistant against man-in-the-middle attacks due to the required physical proximity [116]. Without knowledge of the cryptographic key $K_{\mathcal{M}}$, $\mathcal{A}_{ext}$ cannot generate a valid access token $[T_{\mathcal{L}}]$ for $\mathcal{L}$. Similarly the properties of NFC also protect the authenticity of $\mathcal{M}$ and $\mathcal{C}$ when $\mathcal{M}$ issues a confidentiality-protected token $[T_{\mathcal{L}}]$ to $\mathcal{C}$. Without access to the key $K_{\mathcal{C}}$ stored in the token $[T_{\mathcal{L}}]$ $\mathcal{A}_{ext}$ is unable to deploy modules on $\mathcal{H}$. At runtime, $\mathcal{C}$ and $I_{\mathcal{C}}$ communicate through an authenticated and end-to-end encrypted channel. These properties are bootstrapped from the access token $[T_{\mathcal{L}}]$ issued to $\mathcal{C}$ by $\mathcal{M}$, which prevents $\mathcal{A}_{ext}$ from communicating with the deployed instance.

As noted in Section 6.4.1 either a client $\mathcal{C}$ or a host $\mathcal{H}$ can act as an *internal adversary* $\mathcal{A}_{int}$. On one hand, $\mathcal{H}$ has to be protected from a malicious module deployed by $\mathcal{C}$. To this end, we designed an access control model that mediates which modules M may access sensitive resources R on $\mathcal{H}$. To implement this model the host operating system needs to run modules M deployed in $\mathcal{C}$'s instance $I_{\mathcal{C}}$ in an isolated least privilege container. Our Android-based implementation uses the default UID-based sandboxing mechanism (see Section 6.4.2), which effectively prevents modules M from accessing sensitive resources R directly. Instead, Xapp modules M use the Resource Controller RC as a deputy who enforces the access control policy defined by the manager $\mathcal{M}$. It should be noted that the Resource Controller RC itself thus is an interesting target for attackers due to its highly-privileged status. Access control on its interfaces to Xapp modules M as well as other applications is critical to prevent the introduction of new confused deputy vulnerabilities. To address this concern the access control policy is protected by our token-based authentication and access control scheme, which ensures that it cannot be forged or modified by an internal adversary $\mathcal{A}_{int}$. Dynamic access control queries evaluated by $\mathcal{M}$ at runtime are protected against impersonation, modification and replay by message authentication codes with nonces.

It should be noted that to implement this access control model we rely on the integrity and security of (system) software on the host (see Section 6.4.1). This requirement is inherent to solutions that operate purely on the application layer. Obviously the internal adversary $\mathcal{A}_{int}$ could extend its privileges at runtime if he could compromise any privileged system services. Furthermore, access control solutions based on application-layer deputies, such as Xapp, cannot provide resilience against confused deputy or collusion attacks. For example, malicious modules deployed by different stakeholders could combine their priv-

ileges and use inter-process communication (IPC) to exchange sensitive assets. Reliable control on IPC would require an extension of the underlying operating system, for example based on the FlaskDroid or Android Security Modules architectures discussed in this dissertation (see Chapters 4 and 5). While this is possible, it would not conform to our interoperability requirement. Moreover, we note that attacks using side channels are out of scope of our framework, and stress that these limitations apply to manually deployed applications as well.

On the other hand, sensitive resources R of a client $\mathcal{C}$ need to be protected from a malicious host $\mathcal{H}$. Xapp's module system encourages developers to enclose sensitive operations in separate modules. A client can decide where these modules are executed. Thus Xapp allows clients to ensure that modules accessing or storing sensitive data, such as long-term credentials or contact information, remain on their trusted devices. Of course the adversary could still exploit software errors, hidden backdoors or bad application design, but this risk is not higher than for traditional applications.

6.4.5 *Conclusion*

Computing in personal, commercial and industrial environments is undergoing a paradigm shift. The advent of the Internet of Things (IoT) enables new use cases, in which classical computing platforms, smartphones and tablets, wearables and further electronic devices operate in concert. Application lifecycle management and secure resource sharing become increasingly important aspects in this area.

In this section, we demonstrated how the Xapp framework for smart and secure cross-device IoT applications for Android can address these challenges. With Xapp, Android apps can run distributed on different devices without the need to install them manually on each device. By using an application-layer deputy our Xapp architecture provides fine-grained access control on shared resources without introducing changes to the underlying operating system. Our proof-of-concept implementation for a video call use case demonstrates the practical feasibility of our approach, and we plan to extend our work in several directions, such as prototyping additional use cases and analyzing the feasibility of integrating (semi)automatic code-partitioning techniques to provide flexible tools to developers.

7 DISCUSSION AND CONCLUSION

7.1 DISSERTATION SUMMARY

In this dissertation, we have discussed extensible system-centric access control architectures for smart mobile devices by example of the Android operating system. After a thorough analysis of security aspects of the Android operating system in Chapter 2 we derived requirements for extensible system-centric access control architectures in Chapter 3. To do so, we first defined our adversary model and discussed related work on both offensive as well as defensive mobile security research. Our analysis revealed that while the existing system-centric security extensions aim to achieve diverse goals, many of them actually place similar access control enforcement hooks into all layers of the underlying operating system. This observation motivated our goal of providing generic and extensible security architectures for augmenting Android's permission-based access control model without introducing further changes to the operating system.

Extensible Policy-Driven Access Control on Android. In Chapter 4, we introduced a policy-driven approach towards extensible access control, which extends SELinux kernel-layer type enforcement [153] to the middleware- and application-layer. Our FlaskDroid architecture [34] serves as an efficient basis for the *policy-driven* instantiation of use-case specific security solutions, which we demonstrated by prototyping both novel security extensions as well as prior work. By analyzing both performance impact and policy complexity we have shown that Android is a suitable target for fine-grained access control using type enforcement. The main reason is that operating systems for smart mobile devices, such as Android, provide a rich set of high-level system services via well-defined interfaces, which facilitates the integration of fine-grained access control mechanisms.

Modular Programmable Access Control on Android. In Chapter 5, we applied lessons learned from extensible operating system security research [278] to Android by proposing a *programmatically* extensible access control architecture. Our Android Security Modules (ASM) framework [119] demonstrates the feasibility of mediating access control on *all* layers of the operating system via a unified middleware-layer interface. Our evaluation has shown that both energy consumption and performance overhead can be minimized by activating access control hooks only when they are actually required by the deployed security modules. We further instantiated both novel use-cases as well as prior work using ASM to demonstrate the flexibility of our solution.

Practical Use-Cases. Chapter 6 introduced a set of distinct use-cases which we implemented using system-centric access control architectures. ConXsense (see Section 6.1) prototypes context-driven access control on Android. Our architecture demonstrates that both usability and user privacy can be improved by dynamically configuring security settings on the device based on the usage context. We instantiated ConXsense on both FlaskDroid

and ASM to showcase fundamental differences between policy-driven and system-centric access control by example.

The DroidAuditor architecture [120] introduced in Section 6.2 utilizes the Android Security Modules framework for dynamic application behavior analysis. DroidAuditor monitors application behavior on end-user devices and organizes these observations in a graph-based database. By example of application-layer privilege escalation attacks as well as malicious spyware applications we have shown that our approach enables both end-users and security analysts to intuitively gain insights into application behavior. Our generic graph-based approach can further be augmented with events generated by related work on dynamic program analysis [66].

In Section 6.3, we discussed the BizzTrust architecture [229], which brings security domain isolation to Android using mandatory access control. BizzTrust evolved from our academic TrustDroid prototype [33]. Despite practical concerns regarding the modification of central operating system components we were able to transform this academic Android system security extension into an award-winning [250, 230] and commercially viable product. In this dissertation, we focused on practical challenges that had to be solved during the productization of our initial design. This process mainly concerned mobile device management, network admission control as well as adapting our initial middleware- and kernel-layer access control mechanisms to the peculiarities of recent Android versions.

Finally, it must be acknowledged that despite the favorable security properties of system-centric access control solutions there are situations where their deployment is currently not feasible in practice. The main reason is that it is not always possible to modify the preinstalled operating system. Related work has proposed to address such situations using binary rewriting and inlined reference monitors (IRMs) [70, 16, 289, 54, 53, 199] to shift access control enforcement into untrusted application processes. However, IRM-based approaches cannot provide complete mediation in the presence of native and reflective code. In Section 6.4, we discussed how our Xapp architecture [36] addresses this concern using application-layer deputies [136]. We have evaluated Xapp in an advanced Internet-of-Things scenario, where multiple cooperating but mutually untrusted devices share data and resources. Our evaluation has shown that within their design-inherent limitations application-layer deputies can be a viable alternative to system-centric access control.

7.2 DIRECTIONS FOR FUTURE RESEARCH

Integration of Trusted Computing Technology. Trusted Execution Environments (TEEs) are central parts of hard- and software security architectures of smart mobile devices today. As discussed in Section 2.1.2 these environments are based on highly-privileged CPU modes and hardware memory isolation mechanisms (e.g., ARM TrustZone [7]). Their main goal is the isolated execution and secure storage of sensitive assets outside of the influence of the main operating system (e.g., Android). TEEs separate code and data of multiple stakeholders using containers, denoted *Trustlets*, and we identified two particular use-cases for integrating TEEs with our system-centric access control architectures.

The first use-case concerns device state attestation and network admission control, which are important building blocks for secure enterprise use of smart mobile devices (see Section 6.3.2). The highly-privileged nature of TEEs enables technologies which attest the integrity of the trusted computing base (TCB) towards network infrastructure components. In the context of policy-driven and programmable access control a particularly interesting problem is the efficient and scalable attestation of access control policy enforcement or active security modules. Moreover, within our DroidAuditor architecture (see Section 6.2) TEE technology could be adopted to preserve the integrity and authenticity of application behavior observations despite operating system compromise. Integration of these approaches with network admission control solutions [263] would allow enterprise administrators to restrict access to security-sensitive information stored within the enterprise IT infrastructure to devices that adhere to a defined security state.

The second use case concerns access control on Trustlets. TEEs rely on access control enforcement mechanisms to restrict access to specific Trustlets to authorized applications. The GlobalPlatform Secure Element Access Control specification [91] standardizes these access control mechanisms. It stipulates that access control policies are defined within Trustlets, and that applications communicating with Trustlets are authenticated using public key cryptography. The GlobalPlatform standard relies on a software component within the main operating system (e.g., Android) to enforce these access control policies. Both our FlaskDroid and ASM architectures provide the necessary primitives to implement the functionality of this *Access Control Enforcer (ACE)* component. They not only allow the instantiation of ACE functionality by enforcing corresponding access control rules, but also provide the means to extend the default application authentication procedure with advanced features, for example using context-aware access control [162].

System-centric Access Control and Dynamic Information Flow Analysis. Our system-centric access control architectures operate on the granularity of application processes. On one hand, the process is the designated security boundary of Android applications. On the other hand, this restricted granularity limits the scope of access control enforcement: In their current states, neither FlaskDroid nor ASM can implement fine-grained information flow control due to their limited insights into application processes.

As a first step to address this concern related work has proposed to integrate information flow control into Android [124] based on dynamic taint analysis [66]. The current state of the art however does not adequately address native and reflective software components, which enable adversaries to thwart detection of illegitimate information flows. We thus propose to investigate mechanisms for fine-grained information flow control in the presence of native and reflective code. Addressing this limitation could pave the way for a set of novel use-cases for extensible system-centric access control architectures: An integration of dynamic taint analysis could generally increase the resolution of access control enforcement, which would enable the development of fine-grained information-flow control and analysis solutions based on our designs. For example, the ASM-based DroidAuditor application behavior analysis platform (see Section 6.2) could potentially be extended with more fine-grained information flow information, which would allow security analysts to gain even more detailed insights into application behavior.

8

ABOUT THE AUTHOR

Stephan Heuser is a research assistant at the Technische Universität Darmstadt and the Intel Collaborative Research Institute for Secure Computing (Intel CRI-SC), Germany. He studied computer science at Technische Universität Darmstadt and received his diploma in 2010. Before changing his affiliation to Technische Universität Darmstadt, he worked as a research assistant at Fraunhofer Institute for Secure Information Technology in Darmstadt, Germany. His research focused on security aspects of smart mobile devices in general, and particularly on the design and implementation of flexible and modular access control mechanisms for the Android operating system.

AWARDS

- TeleTrust Innovation Award 2012 (BizzTrust)
- Best Paper Award at ASIACCS 2014 (ConXSense)

PEER-REVIEWED PUBLICATIONS

Stephan Heuser, Marco Negro, Praveen Kumar Pendyala, and Ahmad-Reza Sadeghi. DroidAuditor: Forensic Analysis of Application-Layer Privilege Escalation Attacks on Android. In *Proceedings of the 20th International Conference on Financial Cryptography and Data Security*, FC'16. URL `http://fc16.ifca.ai/preproceedings/15_Heuser.pdf`.

Christoph Busold, Stephan Heuser, Jon Rios, Ahmad-Reza Sadeghi, and N. Asokan. Smart and Secure Cross-Device Apps for the Internet of Advanced Things. In *Proceedings of the 19th International Conference on Financial Cryptography and Data Security*, FC'15. URL `http://dx.doi.org/10.1007/978-3-662-47854-7_17`.

Alexandra Dmitrienko, Stephan Heuser, Thien Duc Nguyen, Marcos da Silva Ramos, Andre Rein, and Ahmad-Reza Sadeghi. Market-driven Code Provisioning to Mobile Secure Hardware. In *Proceedings of the 19th International Conference on Financial Cryptography and Data Security*, FC'15. URL `http://dx.doi.org/10.1007/978-3-662-47854-7_23`.

Stephan Heuser, Adwait Nadkarni, William Enck, and Ahmad-Reza Sadeghi. ASM: A Programmable Interface for Extending Android Security. In *Proceedings of the 23rd USENIX Security Symposium*, USENIX'14. URL `https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/heuser`.

Markus Miettinen, Stephan Heuser, Wiebke Kronz, Ahmad-Reza Sadeghi, and N. Asokan. ConXsense – Context Profiling and Classification for Context-Aware Access Control. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, ASIACCS'14. URL http://dx.doi.org/10.1145/2590296.2590337.

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*, USENIX'13. URL https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/bugiel.

Ammar Alkassar, Stephan Heuser, and Christian Stüble. Vertrauenswürdige Smartphones: Technologien und Lösungen. In *Tagungsband zum 13. Deutschen IT-Sicherheitskongress*. URL https://www.trust.informatik.tu-darmstadt.de/publications/publication-details/?no_cache=1&tx_bibtex_pi1%5Bpub_id%5D=TUD-CS-2013-0142.

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*, SPSM'11. URL http://dx.doi.org/10.1145/2046614.2046624.

Ingo Bente, Gabi Dreo, Bastian Hellmann, Stephan Heuser, Joerg Vieweg, Josef von Helden, and Johannes Westhuis. Towards Permission-Based Attestation for the Android Platform. In *Proceedings of the 4th International Conference on Trust and Trustworthy Computing*, TRUST'11. URL http://dx.doi.org/10.1007/978-3-642-21599-5_8.

Kai-Oliver Detken, Günther Diederich, and Stephan Heuser. Sichere Plattform zur Smartphone-Anbindung auf Basis von TNC. In *D.A.CH Security 2011: Bestandsaufnahme, Konzepte, Anwendungen und Perspektiven*, DACH'11. URL http://www.vogue-project.de/cms/upload/pdf/DACH2011_VOGUE-Beitrag_final.pdf.

Julian Schütte and Stephan Heuser. Auctions for Secure Multi-party Policy Negotiation in Ambient Intelligence. In *Proceedings of the IEEE Workshops of the 25th International Conference on Advanced Information Networking and Applications Workshops*, WAINA'11. URL http://dx.doi.org/10.1109/WAINA.2011.98.

BOOKS

N. Asokan, Lucas Davi, Alexandra Dmitrienko, Stephan Heuser, Kari Kostiainen, Elena Reshetova, and Ahmad-Reza Sadeghi. *Mobile Platform Security*. Morgan & Claypool, 1st edition, 2013. ISBN 1627050973, 9781627050975. URL http://dx.doi.org/10.2200/S00555ED1V01Y201312SPT009.

TECHNICAL REPORTS

Stephan Heuser, Marco Negro, Praveen Kumar Pendyala, and Ahmad-Reza Sadeghi. DroidAuditor: Forensic Analysis of Application-Layer Privilege Escalation Attacks on Android. Technical Report TUD-CS-2016-0025, TU Darmstadt, 2016. URL https://www.trust.informatik.tu-darmstadt.de/research/publications/publication-details/?no_cache=1&tx_bibtex_pi1%5Bpub_id%5D=TUD-CS-2016-0025.

Stephan Heuser, Adwait Nadkarni, William Enck, and Ahmad-Reza Sadeghi. ASM: A Programmable Interface for Extending Android Security. Technical Report TUD-CS-2014-0063, TU Darmstadt, 2014. URL https://www.trust.informatik.tu-darmstadt.de/publications/publication-details/?no_cache=1&tx_bibtex_pi1%5Bpub_id%5D=TUD-CS-2014-0063.

Sven Bugiel, Stephan Heuser, and Ahmad-Reza Sadeghi. Towards a Framework for Android Security Modules: Extending SE Android Type Enforcement to Android Middleware. Technical Report TUD-CS-2012-0231, TU Darmstadt, 2012a. URL https://www.trust.informatik.tu-darmstadt.de/publications/publication-details/?no_cache=1&tx_bibtex_pi1%5Bpub_id%5D=TUD-CS-2012-0231.

Sven Bugiel, Stephan Heuser, and Ahmad-Reza Sadeghi. myTunes: Semantically Linked and User-Centric Fine-Grained Privacy Control on Android. Technical Report TUD-CS-2012-0226, TU Darmstadt, 2012b. URL https://www.trust.informatik.tu-darmstadt.de/publications/publication-details/?no_cache=1&tx_bibtex_pi1%5Bpub_id%5D=TUD-CS-2012-0226.

POSTERS

Stephan Heuser, Bradley Reaves, Praveen Kumar Pendyala, Henry Carter, Alexandra Dmitrienko, William Enck, Ahmad-Reza Sadeghi, and Patrick Traynor. Phonion: Frustrating Telephony Metadata Analysis. In *Proceedings of the 31st Annual Computer Security Applications Conference*, ACSAC'15.

Stephan Heuser, Bradley Reaves, Praveen Kumar Pendyala, Henry Carter, Alexandra Dmitrienko, William Enck, Ahmad-Reza Sadeghi, and Patrick Traynor. Phonion: Frustrating Telephony Metadata Analysis. In *Proceedings of the 22nd Annual Network and Distributed System Security Symposium*, NDSS'15.

Nicolai Kuntze, Stephan Heuser, Carsten Rudolph, Malte Ried, and Michael Jäger. EvidenceCam – Android based approach to provide non-repudiation for digital evidence. In *Proceedings of the 6th IEEE International Workshop on Systematic Approaches to Digital Forensic Engineering*, SADFE'11.

BIBLIOGRAPHY

[1] Yama LSM. URL: https://www.kernel.org/doc/Documentation/security/Yama.txt.

[2] Yousra Aafer, Nan Zhang, Zhongwen Zhang, Xiao Zhang, Kai Chen, XiaoFeng Wang, Xiaoyong Zhou, Wenliang Du, and Michael Grace. Hare Hunting in the Wild Android: A Study on the Threat of Hanging Attribute References. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security*, CCS'15. URL: http://dx.doi.org/10.1145/2810103.2813648.

[3] B. Aboba, L. Blunk, J. Vollbrecht, J. Carlson, and H. Levkowetz. Extensible Authentication Protocol (EAP). RFC 3748, 2008. URL: http://www.ietf.org/rfc/rfc3748.txt.

[4] Vitor Afonso, Antonio Bianchi, Yanick Fratantonio, Christopher Kruegel, Giovanni Vigna, Adam Doupe, and Mario Polino. Going Native: Using a Large-Scale Analysis of Android Apps to Create a Practical Native-Code Sandboxing Policy. In *Proceedings of the 23rd Annual Network and Distributed System Security Symposium*, NDSS'16. URL: https://www.internetsociety.org/sites/default/files/blogs-media/going-native-large-scale-analysis-android-apps-practical-native-code-sandboxing-policy.pdf.

[5] Jeremy Andrus, Christoffer Dall, Alexander Van't Hof, Oren Laadan, and Jason Nieh. Cells: A Virtual Mobile Smartphone Architecture. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP'11. URL: http://dx.doi.org/10.1145/2043556.2043574.

[6] Apple Inc. App Store. URL: https://itunes.apple.com/us/genre/ios/id36?mt=8.

[7] ARM. ARM Security Technology - Building a Secure System using TrustZone Technology, 2009. URL: http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf.

[8] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, and Konrad Rieck. DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket. In *21st Annual Network and Distributed System Security Symposium*, NDSS'14. URL: http://www.internetsociety.org/doc/drebin-effective-and-explainable-detection-android-malware-your-pocket.

[9] 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*, PLDI '14. URL: http://dx.doi.org/10.1145/2594291.2594299.

[10] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. PScout: Analyzing the Android Permission Specification. In *Proceedings of the 19th ACM Conference on Computer and Communications Security*, CCS'12. URL: http://dx.doi.org/10.1145/2382196.2382222.

[11] Vitalii Avdiienko, Konstantin Kuznetsov, Alessandra Gorla, Andreas Zeller, Steven Arzt, Siegfried Rasthofer, and Eric Bodden. Mining Apps for Abnormal Usage of Sensitive Data. In *Proceedings of the 37th IEEE International Conference on Software Engineering*, ICSE'15. URL: http://dx.doi.org/10.1109/ICSE.2015.61.

[12] Golam Sarwar Babil, Olivier Mehani, Roksana Boreli, and Mohamed-Ali Kaafar. On the Effectiveness of Dynamic Taint Analysis for Protecting Against Private Information Leaks on Android-based Devices. In *Proceedings of the 10th International Conference on Security and Cryptography*, SECRYPT'13. URL: http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=7223198.

[13] Michael Backes, Sven Bugiel, and Sebastian Gerling. Scippa: System-Centric IPC Provenance on Android. In *Proceedings of the 30th Annual Computer Security Applications Conference*, ACSAC'14. URL: http://dx.doi.org/10.1145/2664243.2664264.

[14] Michael Backes, Sven Bugiel, Sebastian Gerling, and Philipp von Styp-Rekowsky. Android Security Framework: Extensible Multi-Layered Access Control on Android. In *Proceedings of the 30th Annual Computer Security Applications Conference*, ACSAC'14. URL: http://dx.doi.org/10.1145/2664243.2664265.

[15] Michael Backes, Sven Bugiel, Christian Hammer, Oliver Schranz, and Philipp von Styp-Rekowsky. Boxify: Full-fledged App Sandboxing for Stock Android. In *Proceedings of the 24th USENIX Security Symposium*, USENIX'15. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/backes.

[16] Michael Backes, Sebastian Gerling, Christian Hammer, 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*, TACAS'13. URL: http://dx.doi.org/10.1007/978-3-642-36742-7_39.

[17] Guangdong Bai, Liang Gu, Tao Feng, Yao Guo, and Xiangqun Chen. Context-Aware Usage Control for Android. In *Proceedings of the 6th Iternational ICST Conference on Security and Privacy in Communication Networks*, SecureComm'10. URL: http://dx.doi.org/10.1007/978-3-642-16161-2_19.

[18] Ken Barr, Prashanth Bungale, Stephen Deasy, Viktor Gyuris, Perry Hung, Craig Newell, Harvey Tuch, and Bruno Zoppis. The VMware Mobile Virtualization Platform: Is That a Hypervisor in Your Pocket? *SIGOPS Operating Systems Review*, 44(4), 2010. URL: http://dx.doi.org/10.1145/1899928.1899945.

[19] David Barrera, William Enck, and P.C. van Oorschot. Meteor: Seeding a Security-Enhancing Infrastructure for Multi-market Application Ecosystems. In *Proceedings of the 2012 IEEE Mobile Security Technologies Workshop*, MOST'12. URL: http://mostconf.org/2012/papers/9.pdf.

[20] David Barrera, H. Güneş Kayacik, Paul C. van Oorschot, and Anil Somayaji. A Methodology for Empirical Analysis of Permission-based Security Models and Its Application to Android. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS'10. URL: http://dx.doi.org/10.1145/1866307.1866317.

[21] Mick Bauer. Paranoid Penguin: An Introduction to Novell AppArmor. *Linux Journal*, (148), 2006. URL: http://www.linuxjournal.com/article/9036.

[22] Francis Bea. WhatsApp reads your phone contacts and is breaking privacy laws, 2013. URL: http://www.digitaltrends.com/mobile/whatsapp-breaks-privacy-laws/.

[23] Michael Bell and VItali Lovich. Apparatus and methods for enforcement of policies upon a wireless device. US. Patent 8254902, 2012.

[24] 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*, HotMobile '11. URL: http://dx.doi.org/10.1145/2184489.2184500.

[25] Antonio Bianchi, Jacopo Corbetta, Luca Invernizzi, Yanick Fratantonio, Christopher Kruegel, and Giovanni Vigna. What the App is That? Deception and Countermeasures in the Android User Interface. In *Proceedings of the 36th IEEE Symposium on Security and Privacy*, S&P'15. URL: http://dx.doi.org/10.1109/SP.2015.62.

[26] Antonio Bianchi, Yanick Fratantonio, Christopher Kruegel, and Giovanni Vigna. NJAS: Sandboxing Unmodified Applications in Non-rooted Devices Running Stock Android. In *Proceedings of the 5th ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM'15. URL: http://dx.doi.org/10.1145/2808117.2808122.

[27] Thomas Bläsing, Aubrey-Derrick Schmidt, Leonid Batyuk, Seyit Ahmet Camtepe, and Sahin Albayrak. An Android Application Sandbox System for Suspicious Software Detection. In *Proceedings of the 5th IEEE International Conference on Malicious and Unwanted Software*, MALWARE'10. URL: http://dx.doi.org/10.1109/MALWARE.2010.5665792.

[28] Dan Boneh and Matthew K. Franklin. Identity-Based Encryption from the Weil Pairing. In *Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO'01. URL: http://dx.doi.org/10.1007/3-540-44647-8_13.

[29] David F. C. Brewer and Michael J. Nash. The Chinese Wall Security Policy. In *Proceedings of the 10th IEEE Symposium on Security and Privacy*, S&P'89. URL: http://dx.doi.org/10.1109/SECPRI.1989.36295.

[30] S. Bugiel, S. Heuser, and A.-R. Sadeghi. myTunes: Semantically Linked and User-Centric Fine-Grained Privacy Control on Android. Technical Report TUD-CS-2012-0226, Center for Advanced Security Research Darmstadt, November 2012.

[31] Sven Bugiel. *Establishing Mandatory Access Control on Android OS*. PhD thesis, Saarland University, 2015. URL: http://nbn-resolving.de/urn:nbn:de:bsz:291-scidok-63546.

[32] 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*, NDSS'12. URL: http://www.internetsociety.org/towards-taming-privilege-escalation-attacks-android.

[33] 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*, SPSM'11. URL: http://dx.doi.org/10.1145/2046614.2046624.

[34] 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*, USENIX'13. URL: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/bugiel.

[35] 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*, SPSM'11. URL: http://dx.doi.org/10.1145/2046614.2046619.

[36] Christoph Busold, Stephan Heuser, Jon Rios, Ahmad-Reza Sadeghi, and N. Asokan. Smart and Secure Cross-Device Apps for the Internet of Advanced Things. In *Proceedings of the 19th International Conference on Financial Cryptography and Data Security*, FC'15. URL: http://dx.doi.org/10.1007/978-3-662-47854-7_17.

[37] Liang Cai and Hao Chen. TouchLogger: Inferring Keystrokes on Touch Screen from Smartphone Motion. In *Proceedings of the 6th USENIX Conference on Hot Topics*

in Security, HOTSEC'11. URL: https://www.usenix.org/conference/hotsec11/touchlogger-inferring-keystrokes-touch-screen-smartphone-motion.

[38] Cameron Camp. The BYOD security challenge: How scary is the iPad, tablet, smartphone surge?, February 2012. URL: http://www.welivesecurity.com/2012/02/28/sizing-up-the-byod-security-challenge/.

[39] Yinzhi Cao, Yanick Fratantonio, Antonio Bianchi, Manuel Egele, Christopher Kruegel, Giovanni Vigna, and Yan Chen. EdgeMiner: Automatically Detecting Implicit Control Flow Transitions through the Android Framework. In *Proceedings of the 22nd Annual Network and Distributed System Security Symposium*, NDSS'15. URL: http://www.internetsociety.org/doc/edgeminer-automatically-detecting-implicit-control-flow-transitions-through-android-framework.

[40] Carter Yagemann. Intent Firewall. URL: http://www.cis.syr.edu/~wedu/android/IntentFirewall/.

[41] Patrick P.F. Chan, Lucas Chi Kwong Hui Hui, and Siu-Ming Yiu. DroidChecker: Analyzing Android Applications for Capability Leak. In *Proceedings of the 5th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '12. URL: http://dx.doi.org/10.1145/2185448.2185466.

[42] Kai Chen, Peng Liu, and Yingjun Zhang. Achieving Accuracy and Scalability Simultaneously in Detecting Application Clones on Android Markets. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE'14. URL: http://dx.doi.org/10.1145/2568225.2568286.

[43] Kai Chen, Peng Wang, Yeonjoon Lee, XiaoFeng Wang, Nan Zhang, Heqing Huang, Wei Zou, and Peng Liu. Finding Unknown Malice in 10 Seconds: Mass Vetting for New Threats at the Google-Play Scale. In *Proceedings of the 24th USENIX Security Symposium*, USENIX'15. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/chen-kai.

[44] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing Inter-Application Communication in Android. In *Proceedings of the 9th Annual International ACM Conference on Mobile Systems, Applications, and Services*, MobiSys'11. URL: http://dx.doi.org/10.1145/1999995.2000018.

[45] Mauro Conti, Bruno Crispo, Earlence Fernandes, and Yury Zhauniarovich. CRêPE: A System for Enforcing Fine-Grained Context-Related Policies on Android. *IEEE Transactions on Information Forensics and Security*, 7(5), 2012. URL: http://dx.doi.org/10.1109/TIFS.2012.2204249.

[46] Michael J. Covington, Prahlad Fogla, Zhiyuan Zhan, and Mustaque Ahamad. A Context-Aware Security Architecture for Emerging Applications. In *Proceedings of the 18th Annual Computer Security Applications Conference*, ACSAC'02. URL: http://dx.doi.org/10.1109/CSAC.2002.1176296.

[47] Jonathan Crussell, Clint Gibler, and Hao Chen. Attack of the Clones: Detecting Cloned Applications on Android Markets. In *Proceedings of the 17th European Symposium on Research in Computer Security*, ESORICS'12. URL: http://dx.doi.org/10.1007/978-3-642-33167-1_3.

[48] Jonathan Crussell, Clint Gibler, and Hao Chen. AnDarwin: Scalable Detection of Semantically Similar Android Applications. In *Proceedings of the 18th European Symposium on Research in Computer Security*, ESORICS'13. URL: http://dx.doi.org/10.1007/978-3-642-40203-6_11.

[49] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. MAUI: Making Smartphones Last Longer with Code Offload. In *Proceedings of the 8th Annual International ACM Conference on Mobile Systems, Applications, and Services*, MobiSys'10. URL: http://dx.doi.org/10.1145/1814433.1814441.

[50] Maria Luisa Damiani, Elisa Bertino, Barbara Catania, and Paolo Perlasca. GEO-RBAC: A spatially aware RBAC. *ACM Transactions on Information and System Security*, 10(1), 2007. URL: http://dx.doi.org/10.1145/1210263.1210265.

[51] Lucas Davi, Alexandra Dmitrienko, Christoph Kowalski, and Marcel Winandy. Trusted Virtual Domains on OKL4: Secure Information Sharing on Smartphones. In *Proceedings of the 6th ACM Workshop on Scalable Trusted Computing*, STC'11. URL: http://dx.doi.org/10.1145/2046582.2046592.

[52] 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*, ISC'10. URL: http://dx.doi.org/10.1007/978-3-642-18178-8_30.

[53] Benjamin Davis and Hao Chen. RetroSkeleton: Retrofitting Android Apps. In *Proceeding of the 11th Annual International ACM Conference on Mobile Systems, Applications, and Services*, MobiSys '13. URL: http://dx.doi.org/10.1145/2462456.2464462.

[54] Benjamin Davis, Ben Sanders, Armen Khodaverdian, and Hao Chen. I-ARM-Droid: A Rewriting Framework for In-App Reference Monitors for Android Applications. In *Proceedings of the 2012 IEEE Mobile Security Technologies Workshop*, MOST'12. URL: http://mostconf.org/2012/papers/28.pdf.

[55] Luke Deshotels, Vivek Notani, and Arun Lakhotia. DroidLegacy: Automated Familial Classification of Android Malware. In *Proceedings of the 3rd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*, PPREW'14. URL: http://dx.doi.org/10.1145/2556464.2556467.

[56] Anthony Desnos. Androguard. URL: https://github.com/androguard/androguard.

[57] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, 2008. URL: http://www.ietf.org/rfc/rfc5246.txt.

[58] Michael Dietz, Shashi Shekhar, Yuliy Pisetsky, Anhei Shu, and Dan S. Wallach. Quire: Lightweight Provenance for Smart Phone Operating Systems. In *Proceedings of the 20th USENIX Security Symposium*, USENIX'11. URL: https://www.usenix.org/conference/usenixsecurity11/quire-lightweight-provenance-smart-phone-operating-systems.

[59] Digital Living Network Alliance. Dlna. URL: http://www.dlna.org/.

[60] Alexandra Dmitrienko, Konrad Eriksson, Dirk Kuhlmann, Gianluca Ramunno, Ahmad-Reza Sadeghi, Steffen Schulz, Matthias Schunter, Marcel Winandy, Luigi Catuogno, and Jing Zhan. Trusted Virtual Domains – Design, Implementation and Lessons Learned. In *Proceedings of the 1st International Conference on Trusted Systems*, INTRUST'09. URL: http://dx.doi.org/10.1007/978-3-642-14597-1_10.

[61] Danny Dolev and Andrew C. Yao. On the Security of Public Key Protocols. *IEEE Transactions on Information Theory*, 29(2), 1983. URL: http://dx.doi.org/10.1109/TIT.1983.1056650.

[62] David Dominguez-Sal, P. Urbón-Bayes, Aleix Giménez-Vañó, Sergio Gómez-Villamor, Norbert Martínez-Bazan, and Josep-Lluis Larriba-Pey. Survey of Graph Database Performance on the HPC Scalable Graph Analysis Benchmark. In *Web-Age Information Management: WAIM 2010 International Workshops: IWGD 2010, XMLDM 2010, WCMT 2010, Revised Selected Papers*, IWGD 2010. URL: http://dx.doi.org/10.1007/978-3-642-16720-1_4.

[63] DoMobile Lab. AppLock. URL: https://play.google.com/store/apps/details?id=com.domobile.applock.

[64] Jack Dongarra, Reed Wade, and Paul McMahan. Linpack Benchmark – Java Version. URL: http://www.netlib.org/benchmark/linpackjava/.

[65] Paul Ducklin. The "Stagefright" hole in Android – what you need to know. URL: https://nakedsecurity.sophos.com/2015/07/28/the-stagefright-hole-in-android-what-you-need-to-know/.

[66] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. *ACM Transactions on Computer Systems*, 32(2), 2014. URL: http://dx.doi.org/10.1145/2619091.

[67] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. A Study of Android Application Security. In *Proceedings of the 20th USENIX Security Symposium*, USENIX'11. URL: https://www.usenix.org/conference/usenixsecurity11/study-android-application-security.

[68] William Enck, Machigar Ongtang, and Patrick Mcdaniel. Mitigating Android Software Misuse Before It Happens. Technical report, Pennsylvania State University, 2008. URL: http://www.enck.org/pubs/NAS-TR-0094-2008.pdf.

[69] 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*, CCS '09. URL: http://dx.doi.org/10.1145/1653662.1653691.

[70] Úlfar Erlingsson. *The Inlined Reference Monitor Approach to Security Policy Enforcement*. PhD thesis, Cornell University, 2004. URL: https://ecommons.cornell.edu/handle/1813/5628.

[71] ETH Zürich, Systems Group. jSLP - Java SLP (Service Location Protocol) Implementation. URL: http://jslp.sourceforge.net/.

[72] European Commission. Antitrust: Commission opens formal investigation against Google in relation to Android mobile operating system. URL: http://europa.eu/rapid/press-release_MEMO-15-4782_en.htm.

[73] Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. Why Eve and Mallory Love Android: An Analysis of Android SSL (in)Security. In *Proceedings of the 19th ACM Conference on Computer and Communications Security*, CCS'12. URL: http://dx.doi.org/10.1145/2382196.2382205.

[74] Sascha Fahl, Marian Harbach, Marten Oltrogge, Thomas Muders, and Matthew Smith. Hey, You, Get Off of My Clipboard. In *Proceedings of the 27th International Conference on Financial Cryptography and Data Security*, FC'13. URL: http://dx.doi.org/10.1007/978-3-642-39884-1_12.

[75] Luca Falsina, Yanick Fratantonio, Stefano Zanero, Christopher Kruegel, Giovanni Vigna, and Federico Maggi. Grab'n Run: Secure and Practical Dynamic Code Loading for Android Applications. In *Proceedings of the 31st Annual Computer Security Applications Conference*, ACSAC'15. URL: http://dx.doi.org/10.1145/2818000.2818042.

[76] Federal Trade Commission. Path Social Networking App Settles FTC Charges it Deceived Consumers and Improperly Collected Personal Information from Users' Mobile Address Books, 2013. URL: http://www.ftc.gov/opa/2013/02/path.shtm.

[77] Rafael Fedler, Marcel Kulicke, and Julian Schütte. Native Code Execution Control for Attack Mitigation on Android. In *Proceedings of the 3rd ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM'13. URL: http://dx.doi.org/10.1145/2516760.2516765.

[78] 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*, CCS '11. URL: http://dx.doi.org/10.1145/2046707.2046779.

[79] Adrienne Porter Felt, Serge Egelman, and David Wagner. I've Got 99 Problems, but Vibration Ain'T One: A Survey of Smartphone Users' Concerns. In *Proceedings of the 2nd ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM'12. URL: http://dx.doi.org/10.1145/2381934.2381943.

[80] Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. Android Permissions: User Attention, Comprehension, and Behavior. In *Proceedings of the Eighth Symposium on Usable Privacy and Security*, SOUPS '12. URL: http://dx.doi.org/10.1145/2335356.2335360.

[81] Adrienne Porter Felt, Helen J. Wang, Alexander Moshchuk, Steven Hanna, and Erika Chin. Permission Re-Delegation: Attacks and Defenses. In *Proceedings of the 20th USENIX Security Symposium*, USENIX'11. URL: https://www.usenix.org/conference/usenixsecurity11/permission-re-delegation-attacks-and-defenses.

[82] Yu Feng, Isil Dillig, Saswat Anand, and Alex Aiken. Apposcopy: Automated Detection of Android Malware (Invited Talk). In *Proceedings of the 2nd International Workshop on Software Development Lifecycle for Mobile*, DeMobile 2014. URL: http://dx.doi.org/10.1145/2661694.2661697.

[83] Pietro Ferrara, Omer Tripp, and Marco Pistoia. MorphDroid: Fine-grained Privacy Verification. In *Proceedings of the 31st Annual Computer Security Applications Conference*, ACSAC'15. URL: http://dx.doi.org/10.1145/2818000.2818037.

[84] Adam P. Fuchs, Avik Chaudhuri, and Jeffrey S. Foster. SCanDroid: Automated Security Certification of Android Applications. Technical Report CS-TR-4991, Department of Computer Science, University of Maryland, College Park, 2009. URL: http://spruce.cs.ucr.edu/SCanDroid/papers.html.

[85] FUSE Developers. The reference implementation of the Linux FUSE (Filesystem in Userspace) interface. URL: https://github.com/libfuse/libfuse.

[86] Martin Georgiev, Suman Jana, and Vitaly Shmatikov. Breaking and Fixing Origin-Based Access Control in Hybrid Web/Mobile Application Frameworks. In *Proceedings of the 21st Annual Network and Distributed System Security Symposium*, NDSS'14. URL: http://www.internetsociety.org/doc/breaking-and-fixing-origin-based-access-control-hybrid-webmobile-application-frameworks.

[87] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. AndroidLeaks: Automatically Detecting Potential Privacy Leaks in Android Applications on a Large Scale. In *Proceedings of the 5th International Conference on Trust and Trustworthy Computing*, TRUST'12. URL: http://dx.doi.org/10.1007/978-3-642-30921-2_17.

[88] Clint Gibler, Ryan Stevens, Jonathan Crussell, Hao Chen, Hui Zang, and Heesook Choi. AdRob: Examining the Landscape and Impact of Android Application Plagiarism. In *Proceedings of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys'13. URL: http://dx.doi.org/10.1145/2462456.2464461.

[89] Peter Gilbert, Byung-Gon Chun, Landon P. Cox, and Jaeyeon Jung. Vision: Automated Security Validation of Mobile Apps at App Markets. In *Proceedings of the 2nd International Workshop on Mobile cloud Computing and Services*, MCS'11. URL: http://dx.doi.org/10.1145/1999732.1999740.

[90] Simon Glass. Das U-Boot – the Universal Boot Loader. URL: http://www.denx.de/wiki/U-Boot.

[91] GlobalPlatform Inc. GlobalPlatform Device Technology - Secure Element Access Control. URL: http://www.globalplatform.org/specificationform.asp?fid=7768.

[92] Joel Goncalves, Luis Lino Ferreira, Luis Miguel Pinho, and Guilherme Silva. Handling Mobility on a QoS-Aware Service-based Framework for Mobile Systems. In *Proceedings of the 8th International IEEE/IFIP Conference on Embedded and Ubiquitous Computing*, EUC'10. URL: http://dx.doi.org/10.1109/EUC.2010.24.

[93] Hugo Gonzalez, Andi A. Kadir, Natalia Stakhanova, Abdullah J. Alzahrani, and Ali A. Ghorbani. Exploring Reverse Engineering Symptoms in Android Apps. In *Proceedings of the 8th European Workshop on System Security*, EuroSec'15. URL: http://dx.doi.org/10.1145/2751323.2751330.

[94] Good Technology. Good for Enterprise. URL: https://www.good.com/.

[95] Google Inc. Android Compatibility Test Suite (CTS). URL: https://source.android.com/compatibility/.

[96] Google Inc. Android TV. URL: http://www.android.com/tv/.

[97] Google Inc. ART and Dalvik. URL: https://source.android.com/devices/tech/dalvik/.

[98] Google Inc. Creating and Monitoring Geofences. URL: http://developer.android.com/training/location/geofencing.html.

[99] Google Inc. Pin and unpin screens - Nexus Help. URL: https://support.google.com/nexus/answer/6118421.

[100] Google Inc. Play Store. URL: https://play.google.com/store.

[101] Google Inc. Security-Enhanced Linux in Android. URL: https://source.android.com/security/selinux/.

[102] Google Inc. System Permissions | Android Developers. URL: http://developer.android.com/guide/topics/security/permissions.html#userid.

[103] Google Inc. Verified Boot. URL: https://source.android.com/security/verifiedboot/.

[104] Michael Grace, Yajin Zhou, Qiang Zhang, Shihong Zou, and Xuxian Jiang. RiskRanker: Scalable and Accurate Zero-day Android Malware Detection. In *Proceedings of the 10th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys'12. URL: http://dx.doi.org/10.1145/2307636.2307663.

[105] 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*, WiSec '12. URL: http://dx.doi.org/10.1145/2185448.2185464.

[106] Michael C. Grace, Yajin Zhou, Zhi Wang, and Xuxian Jiang. Systematic Detection of Capability Leaks in Stock Android Smartphones. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium*, NDSS'12. URL: http://www.internetsociety.org/systematic-detection-capability-leaks-stock-android-smartphones.

[107] Kevin Gudeth, Matthew Pirretti, Katrin Hoeper, and Ron Buskey. Delivering Secure Applications on Commercial Mobile Devices: The Case for Bare Metal Hypervisors. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM'11. URL: http://dx.doi.org/10.1145/2046614.2046622.

[108] Aditi Gupta, Markus Miettinen, N. Asokan, and Marcin Nagy. Intuitive Security Policy Configuration in Mobile Devices Using Context Profiling. In *Proceedings of the 2012 ASE/IEE International Conference on Privacy, Security, Risk and Trust, and 2012 ASE International Conference on Social Computing*, PASSAT/SocialCom'12. URL: http://dx.doi.org/10.1109/SocialCom-PASSAT.2012.60.

[109] Joshua D. Guttman, Amy L. Herzog, John D. Ramsdell, and Clement W. Skorupka. Verifying information flow goals in Security-Enhanced Linux. *Journal on Computer Security*, 13(1), 2005. URL: http://dl.acm.org/citation.cfm?id=1066478.

[110] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The WEKA Data Mining Software: An Update. *ACM SIGKDD Explorations Newsletter*, 11(1), 2009. URL: http://dx.doi.org/10.1145/1656274.1656278.

[111] Steve Hanna, Ling Huang, Edward Wu, Saung Li, Charles Chen, and Dawn Song. Juxtapp: A Scalable System for Detecting Code Reuse Among Android Applications. In *Proceedings of the 9th International Conference on Detection of Intrusions and*

Malware, and Vulnerability Assessment, DIMVA'12. URL: http://dx.doi.org/10.1007/978-3-642-37300-8_4.

[112] Hannover University of Applied Sciences and Arts. TNC@FHH | Trust@HsH. URL: http://trust.f4.hs-hannover.de/projects/tncatfhh.html.

[113] Hao Hao, Vicky Singh, and Wenliang Du. On the Effectiveness of API-level Access Control Using Bytecode Rewriting in Android. In *Proceedings of the 8th ACM Symposium on Information, Computer and Communications Security*, ASIACCS'13. URL: http://dx.doi.org/10.1145/2484313.2484317.

[114] Toshiharu Harada, Takashi Horie, and Kazuo Tanaka. Task Oriented Management Obviates Your Onus on Linux. In *Japan Linux Conference*. URL: https://osdn.jp/projects/tomoyo/docs/lc2004-en.pdf.

[115] Norm Hardy. The Confused Deputy: (or Why Capabilities Might Have Been Invented). *ACM SIGOPS Operating Systems Review*, 22(4), 1988. URL: http://dl.acm.org/citation.cfm?id=54289.871709.

[116] Ernst Haselsteiner and Klemens Breitfuß. Security in Near Field Communication. In *Workshop on RFID Security 2006*, RFIDSec'06. URL: http://events.iaik.tugraz.at/RFIDSec06/Program/papers/002%20-%20Security%20in%20NFC.pdf.

[117] Heise Medien. BSI erteilt vorläufige Zulassung für das SecuTABLET. URL: http://heise.de/-3138891.

[118] Gernot Heiser. The Motorola Evoke QA4 - A Case Study in Mobile Virtualization, 2009. URL: https://ssrg.nicta.com.au/publications/papers/Heiser_09:WP:evoke.pdf.

[119] Stephan Heuser, Adwait Nadkarni, William Enck, and Ahmad-Reza Sadeghi. ASM: A Programmable Interface for Extending Android Security. In *Proceedings of the 23rd USENIX Security Symposium*, USENIX'14. URL: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/heuser.

[120] Stephan Heuser, Marco Negro, Praveen Kumar Pendyala, and Ahmad-Reza Sadeghi. DroidAuditor: Forensic Analysis of Application-Layer Privilege Escalation Attacks on Android. In *Proceedings of the 20th International Conference on Financial Cryptography and Data Security*, FC'16. URL: http://fc16.ifca.ai/preproceedings/15_Heuser.pdf.

[121] Tsung-Hsuan Ho, Daniel Dean, Xiaohui Gu, and William Enck. PREC: Practical Root Exploit Containment for Android Devices. In *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*, CODASPY'14. URL: http://dx.doi.org/10.1145/2557547.2557563.

[122] Andrew Hoog. *Android Forensics: Investigation, Analysis and Mobile Security for Google Android*. Syngress Publishing, 1st edition, 2011.

[123] Jann Horn. CVE-2014-7911: Android <5.0 Privilege Escalation using ObjectInputStream. URL: http://seclists.org/fulldisclosure/2014/Nov/51.

[124] 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*, CCS '11. URL: http://dx.doi.org/10.1145/2046707.2046780.

[125] George Hotz. towelroot by geohot. URL: https://towelroot.com/.

[126] Hongxin Hu, Gail-Joon Ahn, and Ketan Kulkarni. Detecting and Resolving Firewall Policy Anomalies. *IEEE Transactions on Dependable and Secure Computing*, 9(3), 2012. URL: http://dx.doi.org/10.1109/TDSC.2012.20.

[127] Joo-Young Hwang, Sang-Bum Suh, Sung-Kwan Heo, Chan-Ju Park, Jae-Min Ryu, Seong-Yeol Park, and Chul-Ryun Kim. Xen on ARM: System Virtualization Using Xen Hypervisor for ARM-Based Secure Mobile Phones. In *Proceedings of the 5th IEEE Consumer Communications and Networking Conference*, CCNC'08. URL: http://dx.doi.org/10.1109/ccnc08.2007.64.

[128] Apple Inc. Use AirPlay to wirelessly stream content from your iPhone, iPad, or iPod touch. URL: https://support.apple.com/en-us/HT204289.

[129] Gartner Inc. Gartner Says Tablets Are the Sweet Spot of BYOD Programs, 2014. URL: http://www.gartner.com/newsroom/id/2909217.

[130] Gartner Inc. Gartner Says Worldwide Smartphone Sales Grew 9.7 Percent in Fourth Quarter of 2015, 2015. URL: http://www.gartner.com/newsroom/id/3215217.

[131] Google Inc. Android. URL: http://www.android.com/.

[132] Google Inc. Android Auto. URL: http://www.android.com/auto/.

[133] Google Inc. Android for Work. URL: https://www.android.com/work/.

[134] Google Inc. Android Interface Definition Language. URL: https://developer.android.com/guide/components/aidl.html.

[135] Google Inc. Android Interfaces and Architecture. URL: https://source.android.com/devices/#HardwareAbstractionLayer.

[136] 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 2nd ACM Workshop on Security and Privacy in Smartphones Mobile Devices*, SPSM'12. URL: http://dx.doi.org/10.1145/2381934.2381938.

[137] Sibei Jiao, Yao Cheng, Lingyun Ying, Purui Su, and Dengguo Feng. A Rapid and Scalable Method for Android Application Repackaging Detection. In *Proceedings of the 11th International Conference on Information Security Practice and Experience*, ISPEC'15. URL: http://dx.doi.org/10.1007/978-3-319-17533-1_24.

[138] Xing Jin, Xuchao Hu, Kailiang Ying, Wenliang Du, Heng Yin, and Gautam Nagesh Peri. Code Injection Attacks on HTML5-based Mobile Apps: Characterization, Detection and Mitigation. In *Proceedings of the 21st ACM Conference on Computer and Communications Security*, CCS'14. URL: http://dx.doi.org/10.1145/2660267.2660275.

[139] Yiming Jing, Ziming Zhao, Gail-Joon Ahn, and Hongxin Hu. Morpheus: Automatically Generating Heuristics to Detect Android Emulators. In *Proceedings of the 30th Annual Computer Security Applications Conference*, ACSAC'14. URL: http://dx.doi.org/10.1145/2664243.2664250.

[140] Graeme Johnson and Michael Dawson. Introduction to Java multitenancy, 2013. URL: http://www.ibm.com/developerworks/java/library/j-multitenant-java/index.html.

[141] Jinyung Kim, Yongho Yoon, Kwangkeun Yi, and Junbum Shin. ScanDal: Static Analyzer for Detecting Privacy Leaks in Android Applications. In *Proceedings of the 2012 IEEE Mobile Security Technologies Workshop*, MOST'12. URL: http://mostconf.org/2012/papers/26.pdf.

[142] William Klieber, Lori Flynn, Amar Bhosale, Limin Jia, and Lujo Bauer. Android Taint Flow Analysis for App Sets. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*, SOAP'14. URL: http://dx.doi.org/10.1145/2614628.2614633.

[143] Sokol Kosta, Andrius Aucinas, Pan Hui, Richard Mortier, and Xinwen Zhang. ThinkAir: Dynamic Resource Allocation and Parallel Execution in the Cloud for Mobile Code Offloading. In *Proceedings of the 31st Annual IEEE International Conference on Computer Communications*, INFOCOM'12. URL: https://dx.doi.org/10.1109/INFCOM.2012.6195845.

[144] Kristen Kennedy and Eric Gustafson and Hao Chen. Quantifying the Effects of Removing Permissions from Android. Applications. In *Proceedings of the 2013 IEEE Mobile Security Technologies Workshop*, MOST'13. URL: http://mostconf.org/2013/papers/25.pdf.

[145] Butler W. Lampson. Protection. *ACM SIGOPS Operating System Review*, 8(1), 1974. URL: http://dx.doi.org/10.1145/775265.775268.

[146] Matthias Lange, Steffen Liebergeld, Adam Lackorzynski, Alexander Warg, and Michael Peter. L4Android: A Generic Operating System Framework for Secure Smartphones. In *Proceedings of the 1st ACM Workshop on Security and Privacy in*

Smartphones and Mobile Devices, SPSM'11. URL: http://dx.doi.org/10.1145/2046614.2046623.

[147] Joseph C. Lehner. AppOpsXposed - AppOps for 4.3+. URL: http://forum.xda-developers.com/xposed/modules/xposed-appopsxposed-appops-4-3-t2564865.

[148] Juanru Li, Yuanyuan Zhang, Wenbo Yang, Junliang Shu, and Dawu Gu. DIAS: Automated Online Analysis for Android Applications. In *Proceedings of the 2014 IEEE International Conference on Computer and Information Technology*, CIT'14. URL: http://dx.doi.org/10.1109/CIT.2014.82.

[149] LIDWIN.PL. Spy Android Phone Let Me Spy - Control your phone online. URL: http://www.letmespy.com/.

[150] Ying-Dar Lin, Yuan-Cheng Lai, Chien-Hung Chen, and Hao-Chuan Tsai. Identifying Android Malicious Repackaged Applications by Thread-grained System Call Sequences. *Computers & Security*, 39, 2013. URL: http://dx.doi.org/10.1016/j.cose.2013.08.010.

[151] Martina Lindorfer, Matthias Neugschwandtner, Lukas Weichselbaum, Yanick Fratantonio, Victor van der Veen, and Christian Platzer. Andrubis - 1,000,000 Apps Later: A View on Current Android Malware Behaviors. In *Proceedings of the 3rd International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security*, BADGERS'14. URL: http://dx.doi.org/10.1109/BADGERS.2014.7.

[152] Anthony Lineberry, David Luke Richardson, and Tim Wyatt. These aren't the Permissions You're Looking for, 2010. URL: http://dtors.files.wordpress.com/2010/08/blackhat-2010-slides.pdf.

[153] Peter Loscocco and Stephen Smalley. Integrating Flexible Support for Security Policies into the Linux Operating System. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, FREENIX'01. URL: https://www.usenix.org/legacy/publications/library/proceedings/usenix01/freenix01/full_papers/loscocco/loscocco_html/index.html.

[154] Robert Love. *Linux Kernel Development*. Addison-Wesley Professional, 3rd edition, 2010.

[155] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities. In *Proceedings of the 19th ACM Conference on Computer and Communications Security*, CCS '12. URL: http://dx.doi.org/10.1145/2382196.2382223.

[156] Tongbo Luo, Hao Hao, Wenliang Du, Yifei Wang, and Heng Yin. Attacks on WebView in the Android System. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11. URL: http://dx.doi.org/10.1145/2076732.2076781.

[157] Christopher Mann and Artem Starostin. A Framework for Static Detection of Privacy Leaks in Android Applications. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12. URL: http://dx.doi.org/10.1145/2245276.2232009.

[158] 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*, ACSAC'12. URL: http://dx.doi.org/10.1145/2420950.2420958.

[159] Philip Marquardt, Arunabh Verma, Henry Carter, and Patrick Traynor. (Sp)iPhone: Decoding Vibrations from Nearby Keyboards Using Mobile Phone Accelerometers. In *18th ACM Conference on Computer and Communications Security*, CCS'11. URL: http://dx.doi.org/10.1145/2046707.2046771.

[160] McAfee. Threats Report May 2015, 2015. URL: http://www.mcafee.com/us/resources/reports/rp-quarterly-threat-q1-2015.pdf.

[161] Patrick McDaniel and Atul Prakash. Methods and Limitations of Security Policy Reconciliation. In *Proceedings of the 23rd IEEE Symposium on Security and Privacy*, S&P'02. URL: http://dx.doi.org/10.1109/SECPRI.2002.1004363.

[162] Markus Miettinen, Stephan Heuser, Wiebke Kronz, Ahmad-Reza Sadeghi, and N. Asokan. ConXsense – Context Profiling and Classification for Context-Aware Access Control. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, ASIACCS'14. URL: http://dx.doi.org/10.1145/2590296.2590337.

[163] André Moulu. Abusing Samsung KNOX to remotely install a malicious application: story of a half patched vulnerability, 2014. URL: http://blog.quarkslab.com/abusing-samsung-knox-to-remotely-install-a-malicious-application-story-of-a-half-patched-vulnerability.html.

[164] André Moulu. Remote Code Execution as System User on Android 5 Samsung Devices abusing WifiCredService (Hotspot 2.0), 2015. URL: http://blog.quarkslab.com/remote-code-execution-as-system-user-on-android-5-samsung-devices-abusing-wificredservice-hotspot-20.html.

[165] Collin Mulliner, Jon Oberheide, William Robertson, and Engin Kirda. PatchDroid: Scalable Third-party Security Patches for Android Devices. In *Proceedings of the 29th Annual Computer Security Applications Conference*, ACSAC '13. URL: http://dx.doi.org/10.1145/2523649.2523679.

[166] Divya Muthukumaran, Joshua Schiffman, Mohamed Hassan, Anuj Sawani, Vikhyath Rao, and Trent Jaeger. Protecting the Integrity of Trusted Applications in Mobile Phone Systems. *Security and Communication Networks*, 4(6), 2011. URL: http://dx.doi.org/10.1002/sec.194.

[167] Yacin Nadji, Jonathon Giffin, and Patrick Traynor. Automated Remote Repair for Mobile Malware. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11. URL: http://dx.doi.org/10.1145/2076732.2076791.

[168] Adwait Nadkarni and William Enck. Preventing Accidental Data Disclosure in Modern Operating Systems. In *Proceedings of the 20th ACM Conference on Computer and Communications Security*, CCS'13. URL: http://dx.doi.org/10.1145/2508859.2516677.

[169] 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*, ASIACCS'10. URL: http://dx.doi.org/10.1145/1755688.1755732.

[170] Neo Technology. Neo4j: The World's Leading Graph Database. URL: http://neo4j.com/company/.

[171] Neo Technology. Neo4j's Graph Query Language: An Introduction to Cypher. URL: http://neo4j.com/developer/cypher-query-language/.

[172] B. Clifford Neuman and Theodore Tso. Kerberos: An Authentication Service for Computer Networks. *IEEE Communications Magazine*, 32(9), 1994. URL: http://dx.doi.org/10.1109/35.312841.

[173] Nils (MWR Security). Building Android Sandcastles in Android's Sandbox, 2010. URL: https://media.blackhat.com/bh-ad-10/Nils/Black-Hat-AD-2010-android-sandcastle-wp.pdf.

[174] Jon Oberheide and Charlie Miller. Dissecting the Android Bouncer. In *SummerCon 2012*. URL: https://jon.oberheide.org/files/summercon12-bouncer.pdf.

[175] Damien Octeau, Somesh Jha, and Patrick McDaniel. Retargeting Android Applications to Java Bytecode. In *Proceedings of the 20th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, FSE'12. URL: http://dx.doi.org/10.1145/2393596.2393600.

[176] Damien Octeau, Daniel Luchaup, Matthew Dering, Somesh Jha, and Patrick McDaniel. Composite Constant Propagation: Application to Android Inter-Component Communication Analysis. In *Proceedings of the 37th IEEE International Conference on Software Engineering*, ICSE'15. URL: http://siis.cse.psu.edu/pubs/octeau-icse15.pdf.

[177] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. Effective Inter-component Communication Mapping in Android with Epicc: An Essential Step Towards Holistic Security Analysis. In *Proceedings of the 22nd USENIX Conference on Security*, USENIX'13. URL: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/octeau.

[178] Department of Defense. *Trusted Computer System Evaluation Criteria*, 1985. URL: http://csrc.nist.gov/publications/history/dod85.pdf.

[179] Machigar Ongtang, Kevin Butler, and Patrick McDaniel. Porscha: Policy Oriented Secure Content Handling in Android. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC'10. URL: http://dx.doi.org/10.1145/1920261.1920295.

[180] Machigar Ongtang, Stephen McLaughlin, William Enck, and Patrick McDaniel. Semantically Rich Application-Centric Security in Android. In *Proceedings of the 25th Annual Computer Security Applications Conference*, ACSAC'09. URL: http://dx.doi.org/10.1109/ACSAC.2009.39.

[181] Oracle. Maxine Virtual Edition - Maxine on Xen. URL: https://kenai.com/projects/guestvm.

[182] Oracle. SecurityManager (Java Platform SE 7). URL: http://docs.oracle.com/javase/7/docs/api/java/lang/SecurityManager.html.

[183] Emmanuel Owusu, Jun Han, Sauvik Das, Adrian Perrig, and Joy Zhang. ACCessory: Password Inference Using Accelerometers on Smartphones. In *Proceedings of the 13th Workshop on Mobile Computing Systems and Applications*, HotMobile'12. URL: http://dx.doi.org/10.1145/2162081.2162095.

[184] Palm Source, Inc. Open Binder. Version 1, 2005. URL: http://www.angryredplanet.com/~hackbod/openbinder/docs/html/index.html.

[185] 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*, ASIACCS'12. URL: http://dx.doi.org/10.1145/2414456.2414498.

[186] Or Peles and Roee Hay. One Class to Rule Them All: 0-Day Deserialization Vulnerabilities in Android. In *Proceedings of the 9th USENIX Workshop on Offensive Technologies*, WOOT'15. URL: https://www.usenix.org/conference/woot15/workshop-program/presentation/peles.

[187] Hao Peng, Chris Gates, Bhaskar Sarma, Ninghui Li, Yuan Qi, Rahul Potharaju, Cristina Nita-Rotaru, and Ian Molloy. Using Probabilistic Generative Models for Ranking Risks of Android Apps. In *Proceedings of the 19th ACM Conference on Computer and Communications Security*, CCS'12. URL: http://dx.doi.org/10.1145/2382196.2382224.

[188] Fabien Perigaud. Local root vulnerability in Android 4.4.2. URL: http://blog.cassidiancybersecurity.com/post/2014/06/Android-4.4.3,-or-fixing-an-old-local-root.

[189] Sebastian Poeplau, Yanick Fratantonio, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications. In *21st Annual Network and Distributed System Security Symposium*. URL: http://www.internetsociety.org/doc/execute-analyzing-unsafe-and-malicious-dynamic-code-loading-android-applications.

[190] Adrienne Porter Felt, Matthew Finifter, Erika Chin, Steven 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*, SPSM '11. URL: http://dx.doi.org/10.1145/2046614.2046618.

[191] Georgios Portokalidis, Philip Homburg, Kostas Anagnostakis, and Herbert Bos. Paranoid Android: Versatile Protection for Smartphones. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC'10. URL: http://dx.doi.org/10.1145/1920261.1920313.

[192] Davy Preuveneers and Yolande Berbers. Context-Driven Migration and Diffusion of Pervasive Services on the OSGi Framework. *International Journal of Autonomous and Adaptive Communications Systems*, 3(1), 2010. URL: http://dx.doi.org/10.1504/IJAACS.2010.030309.

[193] Niels Provos. Improving Host Security with System Call Policies. In *Proceedings of the 12th USENIX Security Symposium*, USENIX'03. URL: https://www.usenix.org/legacy/events/sec03/tech/full_papers/provos/provos_html/index.html.

[194] QNX Software Systems Limited. QNX Operating Systems. URL: http://www.qnx.com/products/neutrino-rtos/index.html.

[195] Qualcomm. Trepn Profiler. URL: https://developer.qualcomm.com/mobile-development/increase-app-performance/trepn-profiler.

[196] Dave Rahardja. Distributing Enterprise Apps. In *Apple Worldwide Developers Conference*, WWDC'14. URL: http://devstreaming.apple.com/videos/wwdc/2014/705xx0r0x0fsaf5/705/705_distributing_enterprise_apps.pdf.

[197] Vikhyath Rao and Trent Jaeger. Dynamic Mandatory Access Control for Multiple Stakeholders. In *Proceedings of the 14th ACM Symposium on Access Control Models and Technologies*, SACMAT'09. URL: http://dx.doi.org/10.1145/1542207.1542217.

[198] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks. In *Proceedings of the 21st Network and Distributed System Security Symposium*, NDSS'14. URL: http://www.internetsociety.org/doc/machine-learning-approach-classifying-and-categorizing-android-sources-and-sinks.

[199] Siegfried Rasthofer, Steven Arzt, Enrico Lovat, and Eric Bodden. DroidForce: Enforcing Complex, Data-centric, System-wide Policies in Android. In *Proceedings of the 9th International Conference on Availability, Reliability and Security*, ARES'14. URL: http://dx.doi.org/10.1109/ARES.2014.13.

[200] Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and Eric Bodden. Harvesting Runtime Values in Android Applications That Feature Anti-Analysis Techniques. In *Proceedings of the 23rd Annual Network and Distributed System Security Symposium*, NDSS'16. URL: https://www.internetsociety.org/sites/default/files/blogs-media/harvesting-runtime-values-android-applications-feature-anti-analysis-techniques.pdf.

[201] Siegfried Rasthofer, Irfan Asrar, Stephan Huber, and Eric Bodden. How Current Android Malware Seeks to Evade Automated Code Analysis. In *Proceedings of the 9th IFIP WG 11.2 International Conference Information Security Theory and Practice*, WISTP'15. URL: http://dx.doi.org/10.1007/978-3-319-24018-3_12.

[202] Vaibhav Rastogi, Yan Chen, and William Enck. AppsPlayground: Automatic Security Analysis of Smartphone Applications. In *Proceedings of the 3rd ACM Conference on Data and Application Security and Privacy*, CODASPY '13. URL: http://dx.doi.org/10.1145/2435349.2435379.

[203] Paul Ratazzi, Ashok Bommisetti, Nian Ji, and Wenliang Du. PINPOINT: Efficient and Effective Resource Isolation for Mobile Security and Privacy. In *Proceedings of the 2015 IEEE Mobile Security Technologies Workshop*, MOST'15. URL: http://www.ieee-security.org/TC/SPW2015/MoST/papers/s3p2.pdf.

[204] Robert W. Reeder, Lujo Bauer, Lorrie Faith Cranor, Michael K. Reiter, and Kami Vaniea. More Than Skin Deep: Measuring Effects of the Underlying Model on Access-control System Usability. In *Proceedings of the ACM CHI Conference on Human Factors in Computing Systems*, CHI'11. URL: http://dx.doi.org/10.1145/1978942.1979243.

[205] Jan S. Rellermeyer, Gustavo Alonso, and Timothy Roscoe. R-OSGi: Distributed Applications Through Software Modularization. In *Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware*, Middleware '07.

[206] Chuangang Ren, Yulong Zhang, Hui Xue, Tao Wei, and Peng Liu. Towards Discovering and Understanding Task Hijacking in Android. In *Proceedings of the 24th USENIX Security Symposium*, USENIX'15. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/ren-chuangang.

[207] Elena Reshetova, Filippo Bonazzi, Thomas Nyman, Ravishankar Borgaonkar, and N. Asokan. Characterizing SEAndroid Policies in the Wild. In *Proceedings of the 2nd International Conference on Information Systems Security and Privacy*, ICISSP'16. URL: http://dx.doi.org/10.5220/0005759204820489.

[208] Franziska Roesner and Tadayoshi Kohno. Securing Embedded User Interfaces: Android and Beyond. In *Proceedings of the 22nd USENIX Security Symposium*, USENIX'13. URL: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/roesner.

[209] Giovanni Russello, Mauro Conti, Bruno Crispo, and Earlence Fernandes. MOSES: Supporting Operation Modes on Smartphones. In *Proceedings of the 17th ACM Symposium on Access Control Models and Technologies*, SACMAT'12. URL: http://dx.doi.org/10.1145/2295136.2295140.

[210] Giovanni Russello, Arturo Blas Jimenez, Habib Naderi, and Wannes van der Mark. FireDroid: Hardening Security in Almost-stock Android. In *Proceedings of the 29th Annual Computer Security Applications Conference*, ACSAC'13. URL: http://dx.doi.org/10.1145/2523649.2523678.

[211] Andrei Sabelfeld and Andrew C. Myers. Language-based Information-flow Security. *IEEE Journal on Selected Areas in Communications*, 21(1), 2006. URL: http://dx.doi.org/10.1109/JSAC.2002.806121.

[212] Norman Sadeh, Jason Hong, Lorrie Cranor, Ian Fette, Patrick Kelley, Madhu Prabaker, and Jinghai Rao. Understanding and Capturing People's Privacy Policies in a People Finder Application. *Personal and Ubiquitous Computing*, 13, 2009. URL: http://dx.doi.org/10.1007/s00779-008-0214-3.

[213] Samsung. AllShare Framework. URL: http://developer.samsung.com/allshare-framework/technical-docs/FAQ.

[214] Samsung. Samsung KNOX. URL: https://www.samsungknox.com.

[215] S. Santesson, M. Myers, R. Ankney, A. Malpani, S. Galperin, and C. Adams. X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP. RFC 6960, 2013. URL: http://www.ietf.org/rfc/rfc6960.txt.

[216] Casey Schaufler. The Smack Project. URL: http://schaufler-ca.com/.

[217] Casey Schaufler. [PATCH v13 0/9] LSM: Multiple concurrent LSMs, 2013. URL: https://lkml.org/lkml/2013/4/23/307.

[218] Roman Schlegel, Kehuan Zhang, Xiao-yong Zhou, Mehool Intwala, Apu Kapadia, and XiaoFeng Wang. Soundcomber: A Stealthy and Context-Aware Sound Trojan for Smartphones. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium*, NDSS'11. URL: http://www.internetsociety.org/doc/soundcomber-stealthy-and-context-aware-sound-trojan-smartphones.

[219] Secure Computing Corporation. DTOS Generalized Security Policy Specification, 1997.

[220] Jaebaek Seo, Daehyeok Kim, Donghyun Cho, Taesoo Kim, and Insik Shin. FlexDroid: Enforcing In-App Privilege Separation in Android. In *Proceedings of*

the 23rd Annual Network and Distributed System Security Symposium, NDSS'16. URL: https://www.internetsociety.org/sites/default/files/blogs-media/flexdroid-enforcing-in-app-privilege-separation-android.pdf.

[221] A. Shabtai, Y. Fledel, and Y. Elovici. Securing Android-Powered Mobile Devices Using SELinux. *IEEE Security Privacy*, 8(3), 2010. URL: http://dx.doi.org/10.1109/MSP.2009.144.

[222] Asaf Shabtai, Uri Kanonov, Yuval Elovici, Chanan Glezer, and Yael Weiss. "Andromaly": a behavioral malware detection framework for android devices. *Journal of Intelligent Information Systems*, 38(1), 2011. URL: http://dx.doi.org/10.1007/s10844-010-0148-x.

[223] Yuru Shao, Xiapu Luo, Chenxiong Qian, Pengfei Zhu, and Lei Zhang. Towards a Scalable Resource-driven Approach for Detecting Repackaged Android Applications. In *Proceedings of the 30th Annual Computer Security Applications Conference*, ACSAC'14. URL: http://dx.doi.org/10.1145/2664243.2664275.

[224] Yuru Shao, Jason Ott, Qi Alfred Chen, Zhiyun Qian, and Zhuoqing Morley Mao. Kratos: Discovering Inconsistent Security Policy Enforcement in the Android Framework. In *Proceedings of the 23rd Annual Network and Distributed System Security Symposium*, NDSS'16. URL: https://www.internetsociety.org/sites/default/files/blogs-media/kratos-discovering-inconsistent-security-policy-enforcement-android-framework_0.pdf.

[225] Shashi Shekhar, Michael Dietz, and Dan S. Wallach. AdSplit: Separating Smartphone Advertising from Applications. In *Proceedings of the 21st USENIX Security Symposium*, USENIX'12. URL: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/shekhar.

[226] Robert Sheldon. MDM puts mobile geofencing, geolocation services on the map. URL: http://searchmobilecomputing.techtarget.com/tip/MDM-puts-mobile-geofencing-geolocation-services-on-the-map.

[227] Robert Siciliano. More Than 30% of People Don't Password Protect Their Mobile Devices, 2013. URL: http://blogs.mcafee.com/consumer/unprotected-mobile-devices.

[228] Laurent Simon and Ross Anderson. PIN Skimmer: Inferring PINs Through the Camera and Microphone. In *Proceedings of the 3rd ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM'13. URL: http://dx.doi.org/10.1145/2516760.2516770.

[229] Sirrix AG security technologies. BizzTrust. URL: https://www.sirrix.com/content/pages/bizztrust_en.htm.

[230] Sirrix AG security technologies. Sirrix wins again the Innovation Price IT - CeBit 2014: a short review. URL: http://www.sirrix.com/content/news/66024.htm.

[231] Sirrix AG security technologies. TrustedObjects Manager. URL: https://www.sirrix.com/content/pages/tom_en.htm.

[232] Stephan 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*, NDSS'13. URL: http://www.internetsociety.org/doc/security-enhanced-se-android-bringing-flexible-mac-android.

[233] Stephen Smalley. SELinux in Android Lollipop and Marshmallow, 2015. URL: http://kernsec.org/files/lss2015/lss2015_selinuxinandroidlollipopandm_smalley.pdf.

[234] Steven Smalley. Middleware MAC for Android, 2012. URL: http://kernsec.org/files/LSS2012-MiddlewareMAC.pdf.

[235] Charlie Soh, Hee Beng Kuan Tan, Yauhen Leanidavich Arnatovich, and Lipo Wang. Detecting Clones in Android Applications Through Analyzing User Interfaces. In *Proceedings of the 23rd IEEE International Conference on Program Comprehension*, ICPC'15. URL: http://dx.doi.org/10.1109/ICPC.2015.25.

[236] Ray Spencer, Stephen Smalley, Peter Loscocco, Mike Hibler, David Andersen, and Jay Lepreau. The Flask Security Architecture: System Support for Diverse Security Policies. In *Proceedings of the 8th USENIX Security Symposium*, USENIX'99. URL: https://www.usenix.org/conference/8th-usenix-security-symposium/flask-security-architecture-system-support-diverse-security.

[237] Michael Spreitzenbarth, Felix Freiling, Florian Echtler, Thomas Schreck, and Johannes Hoffmann. Mobile-sandbox: Having a Deeper Look into Android Applications. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13. URL: http://dx.doi.org/10.1145/2480362.2480701.

[238] SQLite Development Team. SQLite. URL: https://www.sqlite.org.

[239] Frank Stajano. The Resurrecting Duckling – What Next? In *Proceedings of the 8th International Workshop on Security Protocols*, SPW'01. URL: http://dx.doi.org/10.1007/3-540-44810-1_27.

[240] Frank Stajano and Ross J. Anderson. The Resurrecting Duckling: Security Issues for Ad-hoc Wireless Networks. In *Proceedings of the 7th International Workshop on Security Protocols*, SPW'00. URL: http://dx.doi.org/10.1007/10720107_24.

[241] Ryan Stevens, Clint Gibler, Jon Crussell, Jeremy Erickson, and Hao Chen. Investigating User Privacy in Android Ad Libraries. In *Proceedings of the 2012 IEEE Mobile Security Technologies Workshop*, MOST'12. URL: http://mostconf.org/2012/papers/27.pdf.

[242] Mengtao Sun and Gang Tan. NativeGuard: Protecting Android Applications from Third-party Native Libraries. In *Proceedings of the 7th ACM Conference on Security*

and Privacy in Wireless and Mobile Networks, WiSec '14. URL: http://dx.doi.org/10.1145/2627393.2627396.

[243] Mingshen Sun, Mengmeng Li, and John C. S. Lui. DroidEagle: Seamless Detection of Visually Similar Android Apps. In *Proceedings of the 8th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec'15. URL: http://dx.doi.org/10.1145/2766498.2766508.

[244] Mingshen Sun, Min Zheng, John C. S. Lui, and Xuxian Jiang. Design and Implementation of an Android Host-based Intrusion Prevention System. In *Proceedings of the 30th Annual Computer Security Applications Conference*, ACSAC'14. URL: http://dx.doi.org/10.1145/2664243.2664245.

[245] Nathan Sweet. KryoNet. URL: https://github.com/EsotericSoftware/kryonet.

[246] Symantec. Android.Enesoluty. URL: https://www.symantec.com/security_response/writeup.jsp?docid=2012-090607-0807-99.

[247] Symantec. Android.Loozfon | Symantec. URL: http://www.symantec.com/security_response/writeup.jsp?docid=2012-082005-5451-99.

[248] Kimberly Tam, Salahuddin J. Khan, Aristide Fattori, and Lorenzo Cavallaro. CopperDroid: Automatic Reconstruction of Android Malware Behaviors. In *Proceedings of the 22nd Annual Network and Distributed System Security Symposium*, NDSS'15. URL: http://www.internetsociety.org/doc/copperdroid-automatic-reconstruction-android-malware-behaviors.

[249] Andrew S. Tanenbaum and Herbert Bos. *Modern Operating Systems*. Prentice Hall Press, 4th edition, 2014.

[250] TeleTrusT – Bundesverband IT-Sicherheit e.V. ISSE Conference 2012 in Brüssel erfolgreich, TeleTrusT verleiht 'Innovation Award' an Fraunhofer SIT Darmstadt. URL: https://www.teletrust.de/startseite/pressemeldung/?tx_ttnews%5Btt_news%5D=501.

[251] Robert Templeman, Zahid Rahman, David Crandall, and Apu Kapadia. PlaceRaider: Virtual Theft in Physical Spaces with Smartphones. In *Proceedings of the 20th Annual Network and Distributed System Security Symposium*, NDSS'13. URL: http://www.internetsociety.org/doc/placeraider-virtual-theft-physical-spaces-smartphones.

[252] The Apache Software Foundation. Apache Felix. URL: http://felix.apache.org.

[253] The Apache Software Foundation. Apache Felix UPnP. URL: http://felix.apache.org/documentation/subprojects/apache-felix-upnp.html.

[254] The GraphStream Team. GraphStream - A Dynamic Graph Library. URL: http://graphstream-project.org/.

[255] The Linux Foundation. Tizen | An open source, standards-based software platform for multiple device categories. URL: https://www.tizen.org/.

[256] The MITRE Corporation. CVE-2014-3153. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-3153.

[257] The OSGi Alliance. OSGi Service Platform Core Specification Release 4, Version 4.2, 2009. URL: https://osgi.org/download/r4v42/r4.core.pdf.

[258] The UPnP Forum. UPnP Device Architecture 2.0, 2015. URL: http://www.upnp.org/specs/arch/UPnP-arch-DeviceArchitecture-v2.0.pdf.

[259] TheTruthSpy LLC. Mobile spy app, Android spy, Call recording, Whatsapp spy, SMS tracker. URL: http://thetruthspy.com/.

[260] Daniel R. Thomas, Alastair R. Beresford, and Andrew Rice. Security Metrics for the Android Ecosystem. In *Proceedings of the 5th ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM'15. URL: http://dx.doi.org/10.1145/2808117.2808118.

[261] Tresys Technology. SELinux Policy Server. URL: http://oss.tresys.com/projects/policy-server.

[262] Trusted Computing Group. Mobile Trusted Module Specification, 2008. URL: http://www.trustedcomputinggroup.org/files/resource_files/87852F33-1D09-3519-AD0C0F141CC6B10D/Revision_6-tcg-mobile-trusted-module-1_0.pdf.

[263] Trusted Computing Group (TCG). *TNC Architecture for Interoperability, Specification Version 1.5, Revision 4*, 2012. URL: http://www.trustedcomputinggroup.org/files/resource_files/DDBB9EF7-1A4B-B294-D03A12D6C4A8B356/TNC_Architecture_v1_5_r4.pdf.

[264] Trusted Computing Group (TCG). *TNC IF-T Binding to TLS, Specification Version 2.0, Revision 8*, 2013. URL: http://www.trustedcomputinggroup.org/files/resource_files/DDBE9B3E-1A4B-B294-D04A6712D58307CD/TNC_IFT_TLS_v2_0_r8.pdf.

[265] Trusted Computing Group (TCG). *TNC IF-M: TLV Binding Specification Version 1.0 Revision 41*, 2014. URL: http://www.trustedcomputinggroup.org/files/resource_files/DDBC8F4D-1A4B-B294-D0E9E54E0BE5CE7C/TNC_IFM_v1_0_r41-a.pdf.

[266] Trusted Computing Group (TCG). *TNC IF-T: Protocol Bindings for Tunneled EAP Methods, Specification Version 2.0 Revision 5*, 2014. URL: http://www.trustedcomputinggroup.org/files/resource_files/DDBE6A9A-1A4B-B294-D0A22F30F722F7C1/TNC_IFT_EAP_v2_0_r5-a2.pdf.

[267] Trusted Computing Group (TCG). *TNC IF-TNCCS: TLV Binding, Specification Version 2.0 Revision 21*, 2014. URL: http://www.trustedcomputinggroup.org/

files/resource_files/DDBAD766-1A4B-B294-D067C0B081329709/IF-TNCCS_v2_0_r21-a.pdf.

[268] Trustonic. Kinibi Trusted Execution Environment (TEE). URL: https://www.trustonic.com/products/kinibi.

[269] UEFI Forum. UEFI Specifications. URL: http://www.uefi.org/specifications.

[270] Prashant Varanasi and Gernot Heiser. Hardware-supported Virtualization on ARM. In *Proceedings of the 2nd Asia-Pacific Workshop on Systems*, APSys '11. URL: http://dx.doi.org/10.1145/2103799.2103813.

[271] Timothy Vidas and Nicolas Christin. Evading Android Runtime Analysis via Sandbox Detection. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, ASIACCS'14. URL: http://dx.doi.org/10.1145/2590296.2590325.

[272] Timothy Vidas, Jiaqi Tan, Jay Nahata, Chaur Lih Tan, Nicolas Christin, and Patrick Tague. A5: Automated Analysis of Adversarial Android Applications. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM'14. URL: http://dx.doi.org/10.1145/2666620.2666630.

[273] VMWare, Inc. Airwatch. URL: http://www.air-watch.com/.

[274] Ruowen Wang, William Enck, Douglas Reeves, Xinwen Zhang, Peng Ning, Dingbang Xu, Wu Zhou, and Ahmed M. Azab. EASEAndroid: Automatic Policy Analysis and Refinement for Security Enhanced Android via Large-Scale Semi-Supervised Learning. In *Proceedings of the 24th USENIX Security Symposium*, USENIX'15. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/wang-ruowen.

[275] Xueqiang Wang, Kun Sun, Yuewu Wang, and Jiwu Jing. DeepDroid: Dynamically Enforcing Enterprise Policy on Android Devices. In *22nd Annual Network and Distributed System Security Symposium*, NDSS'15. URL: http://www.internetsociety.org/doc/deepdroid-dynamically-enforcing-enterprise-policy-android-devices.

[276] 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*, CODASPY'14. URL: http://dx.doi.org/10.1145/2557547.2557560.

[277] Robert N. M. Watson. TrustedBSD: Adding Trusted Operating System Features to FreeBSD. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, FREENIX'01. URL: https://www.usenix.org/legacy/publications/library/proceedings/usenix01/freenix01/full_papers/watson/watson_html/index.html.

[278] Robert N. M. Watson. A Decade of OS Access-Control Extensibility. *Communications of the ACM*, 56(2), 2013. URL: http://dx.doi.org/10.1145/2408776.2408792.

[279] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps. In *Proceedings of the 2014 ACM Conference on Computer and Communications Security*, CCS '14. URL: http://dx.doi.org/10.1145/2660267.2660357.

[280] Ryan Welton. Remotely Abusing Android, 2015. URL: https://www.blackhat.com/docs/ldn-15/materials/london-15-Welton-Abusing-Android-Apps-And-Gaining-Remote-Code-Execution.pdf.

[281] Primal Wijesekera, Arjun Baokar, Ashkan Hosseini, Serge Egelman, David Wagner, and Konstantin Beznosov. Android Permissions Remystified: A Field Study on Contextual Integrity. In *Proceedings of the 24th USENIX Security Symposium*, USENIX'15. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/wijesekera.

[282] Ryszard Wiśniewski and Connor Tumbleson. android-apktool. URL: http://ibotpeaches.github.io/Apktool/.

[283] Chris Write, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman. Linux Security Modules: General Security Support for the Linux Kernel. In *Proceedings of the 11th USENIX Security Symposium*, USENIX'02. URL: https://www.usenix.org/legacy/events/sec02/full_papers/wright/wright_html/index.html.

[284] Chiachih Wu, Yajin Zhou, Kunal Patel, Zhenkai Liang, and Xuxian Jiang. AirBag: Boosting Smartphone Resistance to Malware Infection. In *Proceedings of the 21st Annual Network and Distributed System Security Symposium*, NDSS'14. URL: http://www.internetsociety.org/doc/airbag-boosting-smartphone-resistance-malware-infection.

[285] Lei Wu, Michael Grace, Yajin Zhou, Chiachih Wu, and Xuxian Jiang. The Impact of Vendor Customizations on Android Security. In *Proceedings of the 20th ACM Conference on Computer and Communications Security*, CCS'13. URL: http://dx.doi.org/10.1145/2508859.2516728.

[286] Luyi Xing, Xiaorui Pan, Rui Wang, Kan Yuan, and XiaoFeng Wang. Upgrading Your Android, Elevating My Malware: Privilege Escalation through Mobile OS Updating. In *Proceedings of the 35th IEEE Symposium on Security and Privacy*, S&P'14. URL: http://dx.doi.org/10.1109/SP.2014.32.

[287] Lei Xu, Guoxi Li, Chuan Li, Weijie Sun, Wenzhi Chen, and Zonhui Wang. Condroid: A Container-Based Virtualization Solution Adapted for Android Devices. In *Proceedings of the 3rd IEEE International Conference on Mobile Cloud Computing,*

Services, and Engineering, Mobilecloud'15. URL: http://dx.doi.org/10.1109/MobileCloud.2015.9.

[288] Nan Xu, Fan Zhang, Yisha Luo, Weijia Jia, Dong Xuan, and Jin Teng. Stealthy Video Capturer: A New Video-based Spyware in 3G Smartphones. In *Proceedings of the 2nd ACM Conference on Wireless Network Security*, WiSec'09. URL: http://dx.doi.org/10.1145/1514274.1514285.

[289] Rubin Xu, Hassen Saidi, and Ross Anderson. Aurasium: Practical Policy Enforcement for Android Applications. In *Proceedings of the 21st USENIX Security Symposium*, USENIX'12. URL: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/xu_rubin.

[290] Zhi Xu, Kun Bai, and Sencun Zhu. TapLogger: Inferring User Inputs on Smartphone Touchscreens Using On-board Motion Sensors. In *Proceedings of the 5th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec'12. URL: http://dx.doi.org/10.1145/2185448.2185465.

[291] 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*, USENIX'12. URL: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/yan.

[292] Zhemin Yang and Min Yang. LeakMiner: Detect Information Leakage on Android with Static Taint Analysis. In *Proceedings of the 2012 Third World Congress on Software Engineering*, WCSE'12. URL: http://dx.doi.org/10.1109/WCSE.2012.26.

[293] 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 20th ACM Conference on Computer and Communications Security*, CCS'13. URL: http://dx.doi.org/10.1145/2508859.2516676.

[294] Fangfang Zhang, Heqing Huang, Sencun Zhu, Dinghao Wu, and Peng Liu. ViewDroid: Towards Obfuscation-resilient Mobile Application Repackaging Detection. In *Proceedings of the 7th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec'14. URL: http://dx.doi.org/10.1145/2627393.2627395.

[295] Mu Zhang and Heng Yin. AppSealer: Automatic Generation of Vulnerability-Specific Patches for Preventing Component Hijacking Attacks in Android Applications. In *Proceedings of the 21st Network and Distributed System Security Symposium*, NDSS'14. URL: http://www.internetsociety.org/doc/appsealer-automatic-generation-vulnerability-specific-patches-preventing-component-hijacking.

[296] Xiao Zhang, Amit Ahlawat, and Wenliang Du. AFrame: Isolating Advertisements from Mobile Applications in Android. In *Proceedings of the 29th Annual Computer*

Security Applications Conference, ACSAC '13. URL: http://dx.doi.org/10.1145/2523649.2523652.

[297] Xiao Zhang, Kailiang Ying, Yousra Aafer, Zhenshen Qiu, and Wenliang Du. Life after App Uninstallation: Are the Data Still Alive? Data Residue Attacks on Android. In *Proceedings of the 23rd Annual Network and Distributed System Security Symposium*, NDSS'16. URL: https://www.internetsociety.org/sites/default/files/blogs-media/life-after-app-installation-data-still-alive-data-residue-attacks-android.pdf.

[298] Xinwen Zhang, Jean-Pierre Seifert, and Onur Acıiçmez. SEIP: Simple and Efficient Integrity Protection for Open Mobile Platforms. In *Proceedings of the 12th International Conference on Information and Communications Security*, ICICS'10. URL: http://dx.doi.org/10.1007/978-3-642-17650-0_9.

[299] Yuan Zhang, Min Yang, Bingquan Xu, Zhemin Yang, Guofei Gu, Peng Ning, X. Sean Wang, and Binyu Zang. Vetting Undesirable Behaviors in Android Apps with Permission Use Analysis. In *Proceedings of the 20th ACM Conference on Computer and Communications Security*, CCS'13. URL: http://dx.doi.org/10.1145/2508859.2516689.

[300] Wu Zhou, Xinwen Zhang, and Xuxian Jiang. AppInk: Watermarking Android Apps for Repackaging Deterrence. In *Proceedings of the 8th ACM Symposium on Information, Computer and Communications Security*, ASIACCS'13. URL: http://dx.doi.org/10.1145/2484313.2484315.

[301] Wu Zhou, Yajin Zhou, Michael Grace, Xuxian Jiang, and Shihong Zou. Fast, Scalable Detection of "Piggybacked" Mobile Applications. In *Proceedings of the 3rd ACM Conference on Data and Application Security and Privacy*, CODASPY'13. URL: http://dx.doi.org/10.1145/2435349.2435377.

[302] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. Detecting Repackaged Smartphone Applications in Third-party Android Marketplaces. In *Proceedings of the 2nd ACM Conference on Data and Application Security and Privacy*, CODASPY'12. URL: http://dx.doi.org/10.1145/2133601.2133640.

[303] Xiaoyong Zhou, Yeonjoon Lee, Nan Zhang, Muhammad Naveed, and XiaoFeng Wang. The Peril of Fragmentation: Security Hazards in Android Device Driver Customizations. In *Proceedings of the 35th IEEE Symposium on Security and Privacy*, S&P'14. URL: http://dx.doi.org/10.1109/SP.2014.33.

[304] Yajin Zhou and Xuxian Jiang. Dissecting Android Malware: Characterization and Evolution. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy*, S&P'12. URL: http://dx.doi.org/10.1109/SP.2012.16.

[305] Yajin Zhou and Xuxian Jiang. Detecting Passive Content Leaks and Pollution in Android Applications. In *Proceedings of the 20th Annual Network and Distributed System Security Symposium*, NDSS'13. URL: http://www.internetsociety.org/doc/detecting-passive-content-leaks-and-pollution-android-applications.

[306] 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*, NDSS'12. URL: http://www.internetsociety.org/hey-you-get-my-market-detecting-malicious-apps-official-and-alternative-android-markets.

[307] Yajin Zhou, Xinwen Zhang, Xuxian Jiang, and Vincent W. Freeh. Taming Information-Stealing Smartphone Applications (on Android). In *Proceedings of the 4th International Conference on Trust and Trustworthy Computing*, TRUST'11. URL: http://dx.doi.org/10.1007/978-3-642-21599-5_7.

[308] Rodrigo ZR. Droidwall. URL: https://play.google.com/store/apps/details?id=com.googlecode.droidwall.free.

Erklärung gemäß §9 der Promotionsordnung

Hiermit versichere ich, die vorliegende Dissertation selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel verfasst zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, Deutschland, Juli 2016

Stephan Heuser