UNIVERSITÄT
DES
SAARLANDES

Saarland University

Faculty of Natural Sciences and Technology I

Department of Computer Science

Establishing Mandatory Access Control on Android OS

Dissertation
zur Erlangung des Grades
des Doktors der Ingenieurwissenschaften
der Naturwissenschaftlich-Technischen Fakultäten
der Universität des Saarlandes

von
Sven Bugiel

Saarbrücken,
August 2015

Tag des Kolloquiums:	14. Dezember 2015
Dekan:	Prof. Dr. Markus Bläser

Prüfungsausschuss:

Vorsitzender:	Prof. Dr. Matteo Maffei
Berichterstattende:	Prof. Dr. Michael Backes
	Prof. Dr. Patrick Drew McDaniel
	Dr. Christian Rossow
Akademischer Mitarbeiter:	Dr. Giancarlo Pellegrino

Zusammenfassung

Gemeinsame Charakteristik aller modernen mobilen Betriebssysteme für sog. ”smart devices” ist eine umfangreiche Diensteschicht, die funktionsreiche Programmierschnittstellen zu der Gerätehardware sowie den Endbenutzerdaten (z.B. Adressbuch) bereitstellt. Um die Systemintegrität, die Privatsphäre des Endbenutzers, sowie die Abgrenzung sich gegenseitig nicht vertrauender Apps effektiv zu gewährleisten, ist es unabdingbar, dass diese Diensteschichten rigide Sicherheitspolitiken umsetzen.

Diese Dissertation präsentiert mehrere Forschungsarbeiten, die “Mandatory Access Control” (MAC) in die Diensteschicht des weit verbreiteten Android Betriebssystems integrieren. Die ersten dieser Arbeiten schufen ein grundlegendes Verständnis für die Integration von Zugriffsmechanismen in das Android Betriebssystem und waren auf sehr spezielle Anwendungsszenarien ausgerichtet. Neuere Arbeiten haben hingegen wichtige Erkenntnisse und Designprinzipien etablierter MAC Architekturen auf herkömmlichen Betriebssystemen für Android adaptiert und mit den speziellen Sicherheitsanforderungen mobiler Systeme verflochten. Die letzte Arbeit in dieser Reihe hat zudem Androids IPC Mechanismus untersucht und dahingehend ergänzt, dass er bessere Informationen über den Ursprung von IPC Nachrichten bereitstellt. Diese Informationen sind fundamental für jedwede Art von Zugriffskontrolle auf Android. Zuletzt diskutiert diese Dissertation aktuelle und zukünftige Forschungsthemen für Zugriffskontrollen auf modernen, mobilen Endgeräten.

Abstract

Common characteristic of all mobile operating systems for smart devices is an extensive middleware that provides a feature-rich API for the onboard sensors and user's data (e.g., contacts). To effectively protect the device's integrity, the user's privacy, and to ensure non-interference between mutually distrusting apps, it is imperative that the middleware enforces rigid security and privacy policies.

This thesis presents a line of work that integrates mandatory access control (MAC) mechanisms into the middleware of the popular, open source Android OS. While our early work established a basic understanding for the integration of enforcement hooks and targeted very specific use-cases, such as multi-persona phones, our most recent works adopt important lessons learned and design patterns from established MAC architectures on commodity systems and intertwine them with the particular security requirements of mobile OS architectures like Android. Our most recent work also complemented the Android IPC mechanism with provisioning of better provenance information on the origins of IPC communication. Such information is a crucial building block for any access control mechanism on Android. Lastly, this dissertation outlines further directions of ongoing and future research on access control on modern mobile operating systems.

Background of this Dissertation

This dissertation is based on the papers mentioned in the following. I contributed to all papers as one of the main authors.

The author had the initial idea and motivation for the TrustDroid [P4] work, i.e., establishing a lightweight domain isolation on Android based on mandatory access control, and defined the fundamental requirement that this access control has to be multi-tiered at the middleware and kernel-level. The author was further responsible for major parts of the design, the implementation, as well as the security discussion and evaluation of TrustDroid. Bhargava Shastry contributed to the TrustDroid work with his port of TOMOYO Linux to Android. Stephan Heuser contributed to this work with his idea of controlling the Linux netfilter from the middleware. Lucas Davi contributed with general writing tasks and was involved in research on related work. Alexandra Dmitrienko was involved in discussions about design decisions. In general all authors performed reviews of the paper.

The author and Alexandra Dmitrienko are the two main co-authors of the XManDroid [P5, T4] work. Alexandra Dmitrienko contributed (with strong support by Lucas Davi), as follow-up to her previous work on privilege escalation attacks [30], the initial idea, attack classification, and conceptual design. She further designed the extension to the VALID policy language and derived example policies for her attack classification. The author contributed to this work several design extensions (Intent tagging, authorization hooks in middleware components, and data filtering by content provider components) and was mainly responsible for the implementation and evaluation (together with Thomas Fischer and Bhargava Shastry). Lucas Davi further contributed a comprehensive study of related work to this work. All authors performed reviews of the paper.

The FlaskDroid [P3, T3, T2] concept was developed in a joint effort between the author, Stephan Heuser, and Ahmad-Reza Sadeghi, with initial contributions from Bhargava Shastry. External input, which influenced the initial idea and consequently also the design, was derived from discussions with Stephen Smalley on SE Android and with N. Asokan on context-aware policies. More specifically, the author was responsible for the requirements analysis and problem description and developing the type enforcement policy language for the Android middleware. The author was, moreover, responsible for major parts of the implementation of the enforcement framework, of some use-cases, and for major parts of the evaluation. Stephan Heuser has several important contributions to the design and implementation, in particular the context provider interface, the domain socket interface of the user space security server, and the extensions to the AIDL compiler. He was further involved in implementing several of the user space object managers and worked on the stabilization of the implementation for the open source release. Stephan Heuser further contributed the attack testbed to the evaluation of FlaskDroid and contributed the implementation of the "Phonebooth mode" and "Privacy-enhanced media store" user-cases as well as parts of the "Secure Logs" use-case.

The author was solely responsible for the requirements analysis, design, implementation, and evaluation of the Android Security Framework [P2, T1]. Philipp von Styp-Rekowsky contributed to this work with his integration of the AppGuard [7] library for inlined authorization hooks into the FlaskDroid architecture as well as implementing the AppGuard use-case. All authors performed reviews of the paper.

The author was exclusively responsible for motivation, requirements analysis, design, implementation, and evaluation of Scippa [P1].

[P1] Backes, M., Bugiel, S., and Gerling, S. Scippa: System-Centric IPC Provenance on Android. In: *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC 2014)*. ACM, 2014.

[P2] Backes, M., Bugiel, S., Gerling, S., and Styp-Rekowsky, P. von. Android Security Framework: Extensible Multi-Layered Access Control on Android. In: *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC 2014)*. ACM, 2014.

[P3] Bugiel, S., Heuser, S., and Sadeghi, A.-R. Flexible and Fine-Grained Mandatory Access Control on Android for Diverse Security and Privacy Policies. In: *Proceedings of the 22nd Usenix Security Symposium (SEC 2013)*. USENIX Association, 2013.

[P4] Bugiel, S., Davi, L., Dmitrienko, A., Heuser, S., Sadeghi, A.-R., and Shastry, B. Practical and Lightweight Domain Isolation on Android. In: *Proceedings of the 1st ACM CCS Workshop on Security and Privacy in Mobile Devices (SPSM 2011)*. ACM, 2011.

[P5] Bugiel, S., Davi, L., Dmitrienko, A., Fischer, T., Sadeghi, A.-R., and Shastry, B. Towards Taming Privilege-Escalation Attacks on Android. In: *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS 2012)*. The Internet Society, 2012.

Further Contributions of the Author

[S1] Backes, M., Bugiel, S., Hammer, C., Schranz, O., and Styp-Rekowsky, P. von. Boxify: Full-fledged App Sandboxing for Stock Android. In: *Proceedings of the 24th USENIX Security Symposium (SEC 2015)*. USENIX Association, 2015.

[S2] Bleikertz, S., Bugiel, S., Ideler, H., Nürnberger, S., and Sadeghi, A.-R. Client-controlled Cryptography-as-a-Service in the Cloud. In: *Proceedings of the 11th International Conference on Applied Cryptography and Network Security (ACNS 2013)*. Springer-Verlag, 2013.

[S3] Bugiel, S., Davi, L., and Schulz, S. Scalable Trust Establishment with Software Reputation. In: *Proceedings of the 6th Annual Workshop on Scalable Trusted Computing (STC 2011)*. ACM, 2011.

[S4] Bugiel, S. and Ekberg, J.-E. Implementing an Application-Specific Credential Platform Using Late-Launched Mobile Trusted Module. In: *Proceedings of the 5th Annual Workshop on Scalable Trusted Computing (STC 2010)*. ACM, 2010.

[S5] Bugiel, S., Dmitrienko, A., Kostiainen, K., Sadeghi, A.-R., and Winandy, M. TruWalletM: Secure Web Authentication on Mobile Platforms. In: *Proceedings of the 2nd Conference on Trusted Systems (INTRUST 2010)*. Springer-Verlag, 2010.

[S6] Bugiel, S., Pöppelmann, T., Nürnberger, S., Sadeghi, A.-R., and Schneider, T. AmazonIA: When Elasticity Snaps Back. In: *Proceedings of the 18th ACM Conference on Computer and Communication Security (CCS 2011)*. ACM, 2011.

[S7] Bugiel, S., Nürnberger, S., Sadeghi, A.-R., and Schneider, T. Twin Clouds: Secure Cloud Computing with Low Latency. In: *Proceedings of the Communications and Multimedia Security Conference (CMS 2011)*. Springer-Verlag, 2011.

[S8] Ekberg, J.-E. and Bugiel, S. Trust in a Small Package: Minimized MRTM Software Implementation for Mobile Secure Environments. In: *Proceedings of the 4th Annual Workshop on Scalable Trusted Computing (STC 2009)*. ACM, 2009.

Technical Reports of the Author

[T1] Backes, M., Bugiel, S., Gerling, S., and Styp-Rekowsky, P. von. *Android Security Framework: Enabling Generic and Extensible Access Control on Android.* Tech. rep. A/01/2014. Saarland University, Apr. 2014.

[T2] Bugiel, S., Heuser, S., and Sadeghi, A.-R. *myTunes: Semantically Linked and User-Centric Fine-Grained Privacy Control on Android.* Tech. rep. TUD-CS-2012-0226. Center for Advanced Security Research Darmstadt, Nov. 2012.

[T3] Bugiel, S., Heuser, S., and Sadeghi, A.-R. *Towards a Framework for Android Security Modules: Extending SE Android Type Enforcement to Android Middleware.* Tech. rep. TUD-CS-2012-0231. Center for Advanced Security Research Darmstadt, Dec. 2012.

[T4] Bugiel, S., Davi, L., Dmitrienko, A., Fischer, T., and Sadeghi, A.-R. *XManDroid: A New Android Evolution to Mitigate Privilege Escalation Attacks.* Tech. rep. TR-2011-04. Technische Universität Darmstadt, 2011.

Acknowledgments

My first thanks go to my current advisor Michael Backes and to my former advisor Ahmad-Reza Sadeghi. Both taught me important lessons for a career in academia and profoundly shaped my development as a PhD student. To Ahmad, I am grateful for offering me my first research position at the System Security Lab at CASED. To Michael, I am very grateful for providing me with the opportunity to continue my PhD studies in the Information Security & Cryptography Group/CISPA at Saarland University after I left CASED. Michael is very passionate about IT security research but also education and it is a great pleasure to work with him and to learn from him. Particularly, I am appreciative that he confided in me from the beginning. Altogether, this created an inspiring and enjoyable work environment, which greatly influenced my decision to stay in academia—something I could not have imagined not all that long ago.

Naturally, many thanks also go to my closest collaborators and co-authors, without whom this thesis would not have been possible and with whom I had really great and fruitful discussions on Android security: Erik Derr, Thomas Fischer, Sebastian Gerling, Stephan Heuser, Bhargava Shastry, and Philipp von Styp-Rekowsky. They made the research work for the papers in this thesis fun and I bonded with them beyond our daily office lives to enjoy together a beer, a round of laser tag, or even playing Mario Kart on the walls of the neighbor's building. Similarly, I want to thank the long list of all other co-authors of my other publications: my very good, long-term friend Sören Bleikertz, Lucas Davi, Jan-Erik Ekberg, Christian Hammer, Hugo Ideler, Kari Kostiainen, Thomas Pöppelmann, Steffen Schulz, Thomas Schneider, Oliver Schranz, and Marcel Winandy. In this sense, I am also thankful to the rest of my (former) colleagues from the System Security Lab and ISC Group for creating such a nice working environment. In particular, I want to give special credits to my (former) colleagues Stephan Heuser, Ünal Kocabaş, Stefan Nürnberger, Oliver Schranz, and Milivoj Simeonovski, with whom I had the pleasure to share an office and who helped me transforming our office into a Nerf gun armory and Mate tea pantry or discovering the nightlife of Saarbrücken. Especially Stefan Nürnberger requires special thanks for taking the step together to leave Darmstadt and to join Michael's group at Saarland University. Special thanks go also to our team assistant Bettina Balthasar, who took so many bureaucratic tasks off my shoulders and whose cheerful nature and wonderful laugh brightened every office day.

I also would like to acknowledge N. Asokan, who gave me the opportunity to work as a research intern in the (unfortunately disbanded) Trustworthy Mobile Platforms group at Nokia Research, and Jan-Erik Ekberg, who instructed me during my internship. Together, they created a wonderful and highly interesting work environment for me and greatly influenced my decision to continue a career in research in the first place.

Additionally, I would like to acknowledge N. Asokan, William Enck, and Stephen Smalley, who reviewed some of the works in this thesis and gave valuable advice and feedback that improved those works.

Moreover, I like to express my gratitude towards Patrick McDaniel and Christian Rossow, who agreed to be the referees of this thesis.

Last but not least, I am deeply grateful to my parents for their unconditional love, their faith in me, and their advice in all situations where I had to leave the ivory tower.

Without them, I would not have been able to go my own way during these last years and would certainly not stand where I stand today.

Contents

1 Introduction **1**

2 Technical Background on Android **7**
- 2.1 Primer on Android . . . 9
  - 2.1.1 Android Software Stack . . . 9
  - 2.1.2 Android Applications . . . 10
- 2.2 Android's Security Philosophy . . . 11

3 TrustDroid **15**
- 3.1 Motivation . . . 17
- 3.2 Problem Description . . . 17
- 3.3 Contributions . . . 19
- 3.4 Technical Problem Description and Model . . . 20
  - 3.4.1 Adversary and Trust Model . . . 20
  - 3.4.2 Objectives and Requirements . . . 21
  - 3.4.3 Assumptions . . . 21
- 3.5 Design of TrustDroid . . . 21
  - 3.5.1 Policy Manager . . . 23
  - 3.5.2 Kernel MAC Manager . . . 26
  - 3.5.3 Firewall Manager . . . 27
- 3.6 Implementation and Evaluation . . . 28
  - 3.6.1 Implementation . . . 28
  - 3.6.2 Evaluation . . . 32
- 3.7 Discussion . . . 32
  - 3.7.1 Security Considerations . . . 32
  - 3.7.2 Trusted Computing . . . 34
- 3.8 Related work . . . 34
  - 3.8.1 Status Quo at Time of Publication . . . 34
  - 3.8.2 Related Work Post-Publication . . . 36
- 3.9 Conclusion . . . 38

4 XManDroid **39**
- 4.1 Motivation . . . 41
- 4.2 Problem Description . . . 41
- 4.3 Our Goal and Contributions . . . 42
- 4.4 Requirements Analysis and Assumptions . . . 43

4.4.1 Adversary Model . . . 43
4.4.2 Objectives and Requirements . . . 43
4.4.3 Assumptions . . . 44
4.5 Framework Architecture . . . 44
4.5.1 Overview . . . 44
4.5.2 Architecture Components . . . 45
4.6 Evaluation . . . 49
4.6.1 Test methodology . . . 50
4.6.2 Study of 3^{rd} Party Application Communication . . . 50
4.6.3 Effectiveness . . . 51
4.6.4 Performance . . . 52
4.6.5 Impact on 3^{rd} party applications' usability . . . 53
4.7 Related Work . . . 54
4.7.1 Status Quo at Time of Publication . . . 54
4.7.2 Related Work Post-Publication . . . 56
4.8 Conclusion . . . 58

5 FlaskDroid 59
5.1 Motivation . . . 61
5.2 Challenges and Our Goal . . . 62
5.3 Contributions . . . 62
5.4 Primer on SELinux and SE Android . . . 63
5.4.1 SELinux . . . 63
5.4.2 SE Android . . . 65
5.5 Requirements Analysis for Android Security Architectures . . . 66
5.5.1 Adversary Model . . . 66
5.5.2 Requirements . . . 67
5.6 FlaskDroid Architecture . . . 69
5.6.1 Overview . . . 69
5.6.2 Architecture Components . . . 70
5.6.3 Policy . . . 77
5.7 Use-Cases / Instantiations . . . 85
5.7.1 Privacy Enhanced System Services and Content Providers . . . 86
5.7.2 Multi-level Security . . . 86
5.7.3 Secure Logs . . . 87
5.7.4 Firewall and Anti-Virus Apps . . . 88
5.7.5 Phone Booth Mode . . . 88
5.7.6 App Developer Policies (Saint) . . . 90
5.8 Evaluation . . . 92
5.8.1 Policy . . . 92
5.8.2 Effectiveness . . . 95
5.8.3 Performance Overhead . . . 100
5.9 Related Work . . . 102
5.9.1 Status Quo at Time of Publication . . . 103
5.9.2 Related Work Post-Publication . . . 106

5.10 Conclusion 108

6 Android Security Framework 109
6.1 Motivation 111
6.2 Problem Description 111
6.3 Contributions 111
6.4 Related Work 113
6.4.1 Extensible Kernel Access Control 113
6.4.2 Android Security 113
6.5 Requirements Analysis 114
6.5.1 Policy as code and not data 114
6.5.2 Policy-agnostic security infrastructure 115
6.6 ASF Architecture 116
6.6.1 Framework Overview 116
6.6.2 Framework Infrastructure 117
6.6.3 Middleware Framework API 119
6.6.4 Middleware Security Modules 122
6.6.5 Stackable and Dynamic Loadable Modules 123
6.7 Example Security Modules 124
6.7.1 AppOps and IntentFirewall 124
6.7.2 CRePE 125
6.7.3 XManDroid 126
6.7.4 Type Enforcement 127
6.7.5 Inlined Reference Monitoring 130
6.7.6 Saint 131
6.7.7 TrustDroid 132
6.7.8 Data shadowing 132
6.7.9 Kirin 133
6.8 Evaluation 133
6.8.1 Performance 133
6.8.2 Current Scope and Future Work 135
6.9 Conclusion 137

7 Scippa 139
7.1 Motivation 141
7.2 Problem Description 141
7.3 Contributions 141
7.4 Binder-Based Inter-App Communication on Android 142
7.4.1 Binder-based ICC 143
7.5 Technical Problem Description 145
7.5.1 Message Dispatching 145
7.5.2 Indirect Communication 146
7.5.3 Provenance Information vs. Permissions 146
7.5.4 Broader Context of this Thesis 147
7.6 Requirements Analysis 148

7.6.1 Adversary Model . . . 148
7.6.2 Requirements and Challenges . . . 148
7.7 System-Centric IPC Call-Chains . . . 149
7.7.1 Establishing Call-Chains . . . 150
7.7.2 Intra-App Call-Chain Propagation . . . 152
7.7.3 Asynchronous Call-Chain Propagation . . . 153
7.7.4 Accessing Call-Chains from User Space . . . 153
7.7.5 Returning Call-Chains to Message Senders . . . 153
7.8 Evaluation . . . 154
7.8.1 Experimental Methodology . . . 154
7.8.2 Performance Impact . . . 154
7.8.3 Binder IPC Provenance . . . 156
7.8.4 Discussion and Limitations . . . 158
7.9 Related Work . . . 159
7.10 Conclusion . . . 161

8 Conclusion **163**

List of Figures

2.1 Background: Android software stack 10
2.2 Background: Android's default security architecture 12

3.1 TrustDroid: Isolation approaches on Android 22
3.2 TrustDroid: TrustDroid architecture 23
3.3 TrustDroid: Coloring and isolation of data 25
3.4 TrustDroid: Coloring of new applications 29

4.1 XManDroid: Framework architecture 45
4.2 XManDroid: Authorization hooks per system component 46
4.3 XManDroid: Example system graph 49
4.4 XManDroid: Visualization of file system access 51
4.5 XManDroid: Visualization of inter-component communication 52

5.1 FlaskDroid: Design concept 69
5.2 FlaskDroid: FlaskDroid architecture 70
5.3 FlaskDroid: SE Android as building block 71
5.4 FlaskDroid: Examples for enforcing default security model 71
5.5 FlaskDroid: CFD of performance overhead on Android v4.0.4 102
5.6 FlaskDroid: CFD of performance overhead on Android v5.1.1 103

6.1 ASF: ASF architecture 116
6.2 ASF: Middleware security module structure 122
6.3 ASF: AppOps and IntentFirewall security module 125
6.4 ASF: CRePE security module 126
6.5 ASF: XManDroid security module 127
6.6 ASF: FlaskDroid security module 128
6.7 ASF: Micro-benchmarks 134
6.8 ASF: Security module performance overhead 136

7.1 Scippa: Binder-based inter-component communication 143
7.2 Scippa: Binder transaction protocol 144
7.3 Scippa: Availability of caller UID 145
7.4 Scippa: Indirection in Intent-based ICC 147
7.5 Scippa: Call-chains during recursive Binder IPC calls 150
7.6 Scippa: Constructing call-chains in recursive transactions 151
7.7 Scippa: Message processing by Handler on Main Thread 152

7.8 Scippa: Performance overhead of Binder transactions 155
7.9 Scippa: CPU cycles overhead for Binder transactions 156
7.10 Scippa: Single Service call roundtrip time vs. payload size 156
7.11 Scippa: Service call roundtrip time vs. call chain length 156
7.12 Scippa: Break down of observed call-chain lengths 157
7.13 Scippa: Call-chain for parallel and ordered broadcasts 159

List of Tables

4.1 XManDroid: ICC timing results 53
4.2 XManDroid: Timing results for system components 53

5.1 FlaskDroid: List of User Space Object Managers 73
5.2 FlaskDroid: Overview of policy complexity 94
5.3 FlaskDroid: Attack testbed scenarios 95
5.4 FlaskDroid: Runtime and memory overhead on Android v4.0.4 101
5.5 FlaskDroid: Performance comparison to related work 101
5.6 FlaskDroid: Runtime overhead on Android v5.1.1 102

6.1 ASF: Break down of hooked system apps/services 120
6.2 ASF: Effort of porting different security extensions as module 124
6.3 ASF: Average performance overhead 133
6.4 ASF: Performance of most frequently invoked hooks 135
6.5 ASF: Security module performance measurements 135

7.1 Scippa: Call-chain statistics 157

List of Code Listings

5.1 FlaskDroid: Reserved UIDs for Android system components 74
5.2 FlaskDroid: Defining middleware-specific objects 77
5.3 FlaskDroid: Assigning types to applications 79
5.4 FlaskDroid: Assigning types to Intents 80
5.5 FlaskDroid: Declaration of contexts 81
5.6 FlaskDroid: Linking booleans with contexts 81
5.7 FlaskDroid: Definition of policy rules 82
5.8 FlaskDroid: Policy language grammar in extended BNF 84
5.9 FlaskDroid: Example policy for securing log entries 88
5.10 FlaskDroid: Example policy for Saint instantiation 90
5.11 FlaskDroid: Example policy for ContactsProvider 97
5.12 FlaskDroid: Example policy for SensorManager 98
5.13 FlaskDroid: Example policy for restricting root user 98
6.1 ASF: Exemplary enforcement functions 118
6.2 ASF: Generic interface to the Linux Security Modules 120
6.3 ASF: Application of generic Kernel module interface 129
6.4 ASF: Generic SELinux LSM interface implementation 129
6.5 ASF: Original FlaskDroid hook in LocationManagerService 130
6.6 ASF: Re-implementation of the FlaskDroid LMS hook 130

1 Introduction

The advent of feature-rich, easily customizable smart devices together with affordable mobile data plans had a profound impact on the ways we produce and consume information today. With smart devices, such as smartphones and tablets, end-users are essentially always-on and continuouly share data easily aggregated using an unprecedented set of features that is bundled into a handheld form factor device. Those features include, for instance, an extensive array of onboard sensors (GPS, motion sensing, camera, etc.), calendar and contacts management, video collections, or various forms of text messaging services. End-users can even further extend their devices' functionality through third party *apps* that are comfortably retrievable from centralized software distribution channels (i.e., *app stores* or *markets*), which seemingly integrate the process of finding and installing new functionality onto the end-users' devices. By today, there is "an app for (almost) everything"—from mobile banking, gaming, increasing productivity, to social networking and more. For Google's Android OS, the by far most widespread mobile operating system according to contemporary market research, are by now more than a million third party apps available on Google's Play app market. The success of this new *app paradigm* also incited the deployment of mobile software stacks in other domains like car systems, smart homes, desktop operating systems, or the Internet of Things, and is accompanied by the ongoing "appification" of those new domains.

Along with the social impact, smart devices also affected the way how developers produce software for those devices. Common characteristic of all major smart device software stacks is that they provide an extensive middleware and application framework on top of the OS, which abstract from the low-level device hardware/features and offer to app developers a feature-rich, extensive API in a high-level programming language. This easily allows app developers to seemingly integrate the device's features (like on-board sensors) and pre-installed services and data management (like user address book or gallery) into their apps.

Smart operating systems have also received very early recognition in the security and privacy research community: The community hinted at the potential privacy risks of feature-rich devices that are tightly integrated in our daily lives, that store a plethora of private information, and that make those information available to a wide spectrum of untrusted app developers through the application framework's API. Research has since dealt with identifying and rectifying privacy and security issues in the smart devices' ecosystem. Although solutions have been introduced that empower the end-user to enhance her privacy protection independently from other actors in the ecosystem (i.e., the developers, market operators, or OS and platform vendors), consensus today is that in the long run effective solutions are a collaborative effort between market mechanisms, such as app vetting, and on-device mechanisms, such as app authentication and access control. Focus of these recent research efforts is the Android OS, which is, in contrast to its walled-garden competitors, provided as open-source.

In this dissertation, we present a line of work that adds mandatory access control to the Android OS and in particular to its extensive middleware layer. Android's architecture incorporated right from its first release many established security best practices and lessons learned. Therefore it offers by design a better protection against malicious applications when compared to classical desktop and server operating systems. However, its early design provided at middleware layer only an application authorization mechanism—*permission* system—that is more akin to a discretionary access control mechanism and, further, it lacked any kind of mandatory access control (MAC) capabilities. This lack of mandatory access control left room for successful attacks against the system integrity and also the user's privacy. Moreover, this lack disqualified Android for deployment in higher-security contexts, such as the increasingly popular dual-usage smartphones for enterprise deployment or the emerging market for governmental mobile devices. Focus of this dissertation was to explore the security design space of historical, established operating systems (e.g., the development of the Linux Security Moduls) and transfer this knowledge to the particular design of the Android OS in order to retrofit Android's design with mandatory access control for enforcing improved security policies that protect the system integrity and the user's private information as well as enable advanced security models on Android. A particular aspect of this work was to explore to which extent the strongly high-level API-oriented design of modern, mobile operating systems provides a better opportunity to more efficiently establish a higher security and privacy standard than is possible on current commodity PC platforms. A particular technical challenge of this line of work was to consider and consolidate enforcement of security policies at both the operating system level (i.e., Linux kernel MAC) and middleware level (i.e., our security extensions to Android).

Our work on establishing mandatory access control on Android stretches across the following peer-reviewed publications [P4, P5, P3, P2, P1], which each contributed to the development of our MAC for Android as presented in this dissertation:

TrustDroid. TrustDroid [P4] is one of the first security extensions to Android that specifically targeted the multi-personna use-case where devices are used in both private and business contexts. This scenario constitutes a classical problem of domain isolation and non-interference between different security domains. However, at the time of this work no effective solution to this problem on Android existed. Prior approaches were based on heavy-weight virtualization or microkernels that are not suitable for deployment on resource-constrained mobile devices. In contrast, TrustDroid introduced a novel mandatory access control architecture for Android's software stack that extended core middleware services and the Linux kernel to enforce static security policies and that isolates private data and apps from business-related data and apps. Moreover, its design included building blocks from the area of Trusting Computing, specifically a Mobile Trusted Module, to enhance the authentication of apps.

XManDroid. XManDroid [P5] is an Android security extension to mitigate the threat of application-level privilege escalation attacks, i.e., collusion or confused deputy attacks. Its design extended that of TrustDroid and introduced a permission-based monitoring system that reflects the observed system state—reduced to the history of performed permission checks and inter-app communication inferred from this history—in a graph-based view. Based on the currently inferred system state, XManDroid's policy enforcement points prohibit any communication that can potentially form a privilege escalation attack. Attack states are described in a policy language based on the VALID policy language for infrastructure cloud environments.

FlaskDroid. FLASKDROID [P3] is the first generic security architecture for the Android OS that can serve as a flexible and effective ecosystem to instantiate different security solutions for the Android software stack. Its design is inspired by the Flask security architecture, where various Object Managers at middleware and kernel-level are responsible for assigning security contexts to their objects and enforce access control on them. A key observation of the FLASKDROID work is, that almost all proposals for Android security extensions in the existing literature (including our TRUSTDROID and XMANDROID) constitute mandatory access control mechanisms that are tailored to the specific semantics of their addressed security problem. By introducing a generic security architecture on top of SE Android together with an efficient policy language (inspired by SELinux) that takes the specifics of Android's middleware semantics into account, FLASKDROID enables a policy-driven instantiation of existing and new security/privacy models. We show the flexibility of our architecture by instantiating selected security models such as existing related work.

Android Security Framework. Although FLASKDROID was the first generic security architecture for Android and was able to instantiate different security models, its design still confined policy authors to the semantics of the FLASKDROID policy language. To remedy this situation, we introduced our ANDROID SECURITY FRAMEWORK (ASF) [P2], a generic, extensible security framework for Android that enables the development and integration of a wide spectrum of security models in form of code-based security modules. The design of ASF reflects lessons learned from the literature on established security frameworks on commodity platforms (such as Linux Security Modules or the BSD MAC Framework) and intertwines them with the particular requirements and challenges from the design of Android's software stack to build a policy-agnostic and multi-tiered security infrastructure. ASF provides a novel security API that supports authors of Android security extensions in developing their modules and supports implementation of policy logic that resembles mandatory result automata. This design overcomes the current unsatisfactory situation to provide security solutions as separate patches to the Android software stack or to embed them into Android's mainline codebase. We demonstrated ASF's effectiveness and efficiency by modularizing different security models from related work, such as dynamic permissions, inlined reference monitoring, and type enforcement.

Scippa. SCIPPA [P1] is an extension to the Android IPC mechanism that establishes inter-process communication (IPC) call-chains across application processes. Thus, although SCIPPA is itself not an access control mechanism, it contributes greatly to the domain of access control on Android by providing essential information: IPC provenance information required to effectively prevent recent attacks. Any kind of enforcement point within the Android middleware relies on such provenance information to identify the subjects in access control decisions. Android's IPC mechanism (Binder) provides provenance information for a one-hop distance. However, it fails in providing sufficient provenance information for all possible inter-app communication scenarios, which has led to a number of attacks. Our solution constitutes a system-centric approach that extends the Binder kernel module and Android's message handlers to provide the needed IPC provenance information. SCIPPA integrates seamlessly into the system architecture.

Outline

The remainder of this dissertation is structured as follows. In Chapter 2, we provide necessary, common technical background information on the Android software stack and its security architecture. We present TrustDroid in Chapter 3 and XManDroid in Chapter 4. FlaskDroid is presented in Chapter 5, our Android Security Framework in Chapter 6, and Scippa in Chapter 7. We conclude this dissertation in Chapter 8.

2
Technical Background on Android

In this chapter we provide common technical background information on Android that is required for better understanding of the following Chapters 3 through 7. Background information that is specific to certain chapters is provided in the respective chapters.

2.1 Primer on Android

Android is an open-source software stack tailored for embedded devices that was released in 2008 [124] by Google as lead of the *Open Handset Alliance*. Android is specifically designed as mobile operating system for touch-enabled devices like smartphones and tablets. In this still rapidly growing market for smartphones, Android has become the dominant mobile operating and held in 2015 a worldwide market share of 80.7% [62, Table 3]. At the time of writing, Android has matured to its version 5.1.1 (code name *Lollipop*) with API version 22 [79].

More recently, Google has extended Android's application domains to new emerging markets for mobile operating systems, such as televisions [81], cars [78], wearables [82], and digital cameras [160]. It even is the foundation for Google's *Project Brillo* [83] for the *Internet of Things*.

2.1.1 Android Software Stack

Android's software stack [17] builds on top of a (slightly modified) Linux kernel and includes a middleware framework and an application layer (as depicted in Figure 2.1). The Linux kernel is responsible for basic operating system services, such as memory management, device drivers, or inter-process communication (IPC), and has been modified by Google to be compatible with the scarce resources of mobile devices, e.g. available power and memory.

On top of the Linux kernel lies the extensive Android middleware. It consists of native libraries (e.g., SSL, WebKit, sqlite), the Android runtime with the *Dalvik Virtual Machine* (DVM), and the application framework. DVM is a register-based virtual machine that is optimized for resource constrained mobile devices and executes Dalvik Executable Bytecode (DEX). With Android version 5.0, Google introduced the *Android Runtime* (ART) as a replacement for DVM. ART, in contrast to DVM, applies ahead-of-time compilation to improve the apps' runtime performance and efficiency at the cost of slightly increased binary size.

The *application framework* of the middleware implements the bulk of Android's application API and provides app developers with a feature-rich API that exposes central platform features such as the onboard sensors array, location services, settings, or GUI management. The application framework is complemented by pre-installed system apps at the application layer, such as *Phone*, *Browser*, or *Contacts*. The API can be extended with 3^{rd} party apps on top of the software stack, which are usually retrieved from centralized software distribution channels such as Google Play or, as allowed by Android's open application model, via side-loading (e.g., from SDCard).

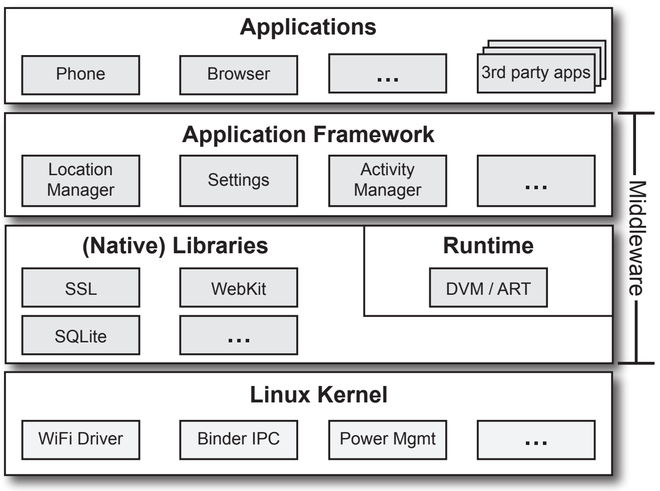

Figure 2.1: Android software stack. (Figure adapted from [17]).

2.1.2 Android Applications

Application layer and application framework apps and services are commonly written in Java and then compiled to *dex* bytecode for execution inside the DVM or ART, respectively. In addition to dex bytecode, apps and services can use (custom) native code libraries (i.e., C/C++) for low-level interactions with the underlying Linux system. Native code can be seamlessly integrated into dex bytecode by means of the *Java Native Interface* (JNI).

All new app processes are by default forked from a designated process called *Zygote*. Zygote is essentially a "warmed up" application runtime that pre-loaded all required core libraries and is just missing the actual application code. By forking the Zygote process and loading application code into the newly spawned process, the application load time during application launch is significantly reduced. As a consequence, every application is executing in its own process.

Android apps are generally composed of different components. The four basic app components are:

Activities: A foreground task that implements a user interface with which the user can interact.

Services: Long-running tasks that usually perform in the background, even when the application is not in the foreground, and that can provide an IPC interface to which other processes can *bind* in order to interact with the *Service* component.

ContentProviders: Encapsulate structured sets of data and manage access to them (by other applications) via an SQLite-like interface.

BroadcastReceivers: Implements a mailbox for broadcast *Intent* messages (see next paragraph).

Inter-Component Communication. Apps can communicate with each other on multiple layers, including standard Linux Inter-Process Communication (IPC) using, e.g., domain sockets, files, or Internet sockets. However, the primary mechanism on Android to connect the components of different application processes is an Android-specific inter-process communication called Binder—a light-weight implementation of OpenBinder [142, 45]—which has simply been coined as *Inter-Component Communication* (ICC) by Enck, Ongtang, and McDaniel [45].

To hide the low-level mechanics of Binder-based ICC from app developers, Android provide different layers of abstraction. Most prominently, *Intent* messages provide an abstract way of performing ICC by describing an operation that should be performed and letting the system deliver this message to an appropriate receiver component(s). Alternatively, the app developer can also explicitly state the receiving component of an *Intent* message. For instance, launching an Activity component is accomplished by sending a suitable *Intent*. Additionally, the Android SDK provides different developer tools to define and bootstrap custom remotely-callable interfaces, e.g., the *Android Interface Definition Language* (AIDL) and AIDL compiler for creating bindable Service components. More technical details on ICC are provided in Chapter 7.

2.2 Android's Security Philosophy

Since Android's introduction, Google has constantly extended and improved Android's security features across its different versions [73, 74, 75, 76]. Today, Android'd security concept includes well-known techniques to mitigate software exploitation (e.g., Address Space Layout Randomization or fortification of code), applies Linux capabilities to reduce the attack surfaces of critical system processes, supports certificate pinning for SSL/TLS connections, or provides various security management aspects such as MDM, application verification, or multi-user support.

However, the cornerstone of Android's security architecture right from its first public release—and also the focus of the line of work presented in this thesis—is application sandboxing and privilege separation between apps.

Application Sandboxing. Android's security philosophy dictates that all apps are sandboxed by executing them in separate processes (by means of forking them from *Zygote*) with distinct user IDs (*UID*). These UIDs are assigned to 3^{rd} party apps during their installation and are taken from the pool of available Linux UIDs in the system. Thus, Android essentially applies Linux' mechanism to separate different *human* users (and their processes) from each other to separate different *apps* (and their processes) from each other. Along this line, each installed app also owns a private directory on the file system based on its UID. In addition, Android reserves a number of well-known UIDs for system services and apps. Moreover, app developers may decide that multiple of their apps may use the same UID (*shared UID*) and hence share the same sandbox. However, this requires that all apps in a shared sandbox come from the same origin, identified through their developer signature.

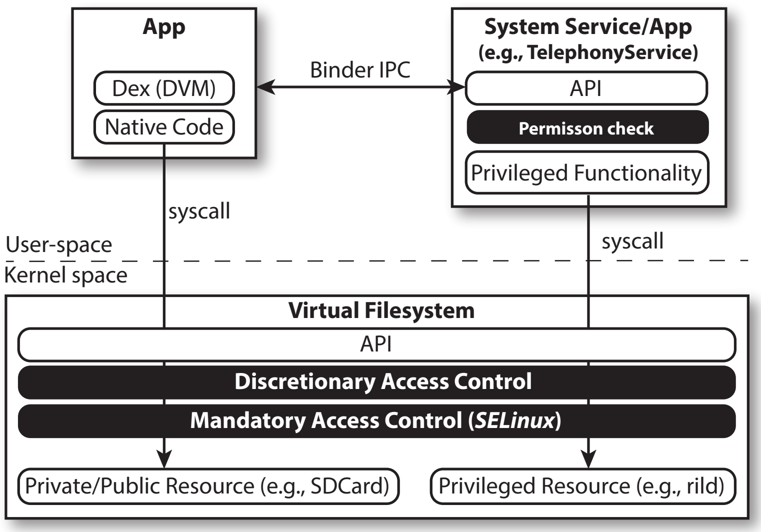

Figure 2.2: Android's default security architecture.

Privilege Separation. To achieve privilege separation between apps, Android introduces *Permissions*, i.e., privileges that an app can request and which are granted by the user at install-time. Currently, the user must grant all requested permissions or abort installation of the app. All granted permissions are immutably assigned to the app's UID.[1] For Android API level 21, more than 150 permissions are defined. The majority of these permissions are simply String labels that are assigned to the app's UID and that govern how the app's processes can interact with other apps' components via ICC. However, a small number of permissions are used to restrict access to some low level resources, such as external storage (e.g. a MicroSD card) or network access. Those permissions are mapped to Linux group identifiers (GIDs) assigned to an app's UID during installation when the respective permission was granted by the user.

In accordance with the least privilege principle, an app without permissions is not able to access security and privacy sensitive resources. Permissions are enforced at two different points in the system architecture (cf. Figure 2.2): First, every app sandbox can directly interact with the kernel through system calls, for instance, to edit files or open a network socket.[2] These resources are either of private nature (i.e., are within the app's private directory) or public resources (e.g., SDCard). Access control in the filesystem ensures that the apps' processes have the necessary rights (i.e., permissions) to issue particular syscalls, e.g., opening a file. The filesystem access control consists of the traditional Linux Discretionary Access Control (DAC), which is complemented (since Android v4.3) by SELinux based Mandatory Access Control (MAC). More information on SELinux and its integration into Android's software stack is provided at the beginning of Chapter 5.

[1]With the upcoming Android version M, Google supposedly overhauls the permission framework and introduces revocable permissions.

[2]It is noteworthy, that the Dalvik VM is, in contrast to the Java VM, **not** a security boundary, since Android explicitly allows applications to include native code that executes outside the DVM.

Second, apps can interact through the Android API in a strictly controlled manner with highly privileged resources. To ensure system security and stability, apps are prohibited to access those resources directly. Instead, those resources are wrapped (or owned) by system services and apps that implement the Android application API. For instance, the *TelephonyService* communicates on behalf of apps with the radio interface layer daemon (*rild*) to initiate calls or send text messages. Whether an app is sufficiently privileged to successfully call the API is determined by a permission check within the system services/apps. For this check, the Binder IPC mechanism provides to the callee (system service/app) the UID of the caller (app).

It should be noted that the permission model is *not* mandatory access control, since callees must discretely deploy or define the required permission check and, moreover, permissions can be freely delegated (e.g., URI permissions [77]).

3 TrustDroid

Practical and Lightweight Domain Isolation on Android

3.1 Motivation

Sophisticated mobile devices, such as smartphones, have become an integral part of our daily life and form for many consumers the digital communication hub with the rest of the world. Not surprisingly, their combination of mobility, connectivity, and versatility has also made them an attractive tool for conducting business transactions: They provide employees a means to remain always connected to the company's network, thereby enabling *on the road* access to company's data through a large variety of mobile office apps. For instance, they allow employees to read and send e-mails, synchronize calendars, organize meetings, create and edit documents, attend telephone and video conferences, and much more.

However, corporations have higher security requirements for the protection of their assets than the average mobile user. Those requirements include, for instance, device management features and, equally important, strict isolation of corporate assets from private assets as well as ensuring non-interference between corporate and private processes. To effectively and securely enable the desired *dual-usage* phone for work and private purposes at the same time, it is imperative that the mobile platform satisfies the corporate security requirements.

3.2 Problem Description

Google has reacted to the ongoing trend of using (Android-based) mobile devices for private as well as work purposes by gradually extending Android's domain isolation and management features. Already early in Android's version history it introduced first simple device management features [69]. However, until very recently the situation for providing a strong security boundary between corporate and private data on Android was very bleak. Only with Android version 5.0 Google introduced new business-focused features [80] that include work-specific profiles tho isolate work data (based on the integration of Samsung's Knox technology [116]). At the time of conducting the work presented in this chapter, however, Android did not fulfill the required security requirements for deploying corporate assets securely on Android-based mobile end-devices. Three major problems had to be faced for enhancing Android's security framework at that time: preventing malicious apps from stealing data or (easily) escalating their privileges, establishing domain isolation between different security domains, and design considerations for the constrained resources of mobile platforms (i.e., battery lifetime or memory capacity).

Security Deficiencies of Android. The core security mechanisms of Android are *application sandboxing* and a *permission framework* (cf. Chapter 2.2 on Android's security architecture). However, different attacks have demonstrated that Android's security architecture is vulnerable to *malware*. First, it has been shown that malicious apps are distributed via different channels including the official Android Market (today Play) and can hence potentially infect end-user devices through sources that are generally considered more trusted [135, 113, 16]. Contributing factor to this is, that Google did at the time of writing not perform code inspection or other advanced application vetting techniques, but just rudimentary automated, dynamic analysis based on the Android emulator (a solution codenamed *Bouncer*). Further, malicious apps may mount application-level privilege escalation attacks that allow an adversary to perform unauthorized actions. This can be achieved by exploiting a vulnerable deputy application [44, 30, 148], or through malicious colluding applications [161, P5]. In particular, privilege escalation attacks have been utilized to send unauthorized text messages [30], to trigger malicious downloads [111, 135], to change the WiFi settings [148], or to perform context-aware voice recording [161]. Finally, any attack at the kernel-level (e.g., file ystem) will allow the adversary to circumvent solutions that only consider security enhancements at the application framework level.

Domain Isolation on Default Android. In the light of recent attacks, the default Android OS at that time could not meet the security requirements of the business world. These requirements mainly encompass the protection of corporate data and applications on the phone along with the security of a heterogeneous company network to which the employees' smartphones connect. In particular, Android lacked data isolation: For instance, standard Android only provides single database instances for *SMS*, *Calendar*, and *Contacts* information. Thus, corporate and private data are stored by default in the same databases and any application with access to the database has immediate access to corporate and private information. Apart from application sandboxing, Android provides no means to *isolate* corporate applications from private user applications in a system-centric way. Hence, an adversary could get unauthorized access to corporate apps (and their data) by utilizing privilege escalation attack techniques. Finally, Android fails to enforce isolation at the network-level, including a lack of context-sensitivity. For instance, there is no means to deny Internet access for untrusted applications while the employee is connected to the company's network, and thus infected employee smartphones pose an additional risk to the corporate network.

To summarize, default Android has no means to group applications and data into *domains*, where in our context a domain compromises a set of applications and data belonging to one trust level (e.g., private, academic, enterprise, department, institution, etc.)

Existing Security Extensions to Android. At that time this work was conducted, a number of security extensions for Android had been proposed, the closest to our work being [140, 46, 139, 134, T4]. However, as we will elaborate in detail in related work (see Section 3.8), all of these solutions focus on a specific layer of the Android software stack (mainly Android's middleware) and fail if the attack occurs on a different layer, e.g., at the network layer by mounting a privilege escalation attack over socket connections [30]). Specifically, they do not address OS-level attacks [136, 111] through which an adversary can, for instance, gain access to the entire file system. Attacks on the OS-level could be mitigated by enabling SELinux on Android [166], however, SELinux operates at the OS-level and lacks the necessary semantics to address attacks at the Android's middleware layer.

In particular, we were not aware of any security extension providing efficient and scalable application and data isolation at different layers of the Android software stack, which is essential for deploying Android in a business context.

On the other hand, several virtualization-based approaches aim at providing domain isolation on Android [11, 108, 31, 4] and can be used for separating private and corporate data/apps on Android. However, contemporary mobile virtualization solutions suffer from practical deficiencies (as we will discuss in Section 3.8): (1) they do not scale well on resource-constrained smartphone platforms that allow only a limited number of virtual machines to be executed simultaneously, in particular in light of the not yet widespread hardware support for mobile virtualization; (2) more importantly, virtualization duplicates the whole Android operating system including security-irrelevant code and thus negatively affects system performance including the battery life-time. This raises a severe usability problem.

3.3 Contributions

In this work, we present a novel security architecture, called TrustDroid, that enables *practical and lightweight domain isolation on each layer* of the Android software stack. Specifically, TrustDroid provides application and data isolation by controlling the main communication channels in Android, namely IPC (Inter-Process Communication), files, databases, and socket connections. TrustDroid is lightweight in the sense that it has a low computational overhead and requires no duplication of Android's middleware and kernel code, which is typically a must for virtualization-based approaches [11, 108]. As a benefit, TrustDroid offers a good scalability in terms of the number of parallel existing domains. In particular, TrustDroid exploits coloring of separate and distinguishable components (this approach has its origins in information-flow theory [155]). We color applications and user data (stored in shared databases) based on a (lightweight) certification scheme which can be easily integrated (as we shall show) into Android. Based on the applications colors, TrustDroid organizes applications along with their data into logical domains. At runtime, TrustDroid monitors all application communication, access to common shared databases, as well as file system and network access, and denies any data exchange or application communication between different domains. In particular, our framework provides the following features:

Mediating IPC: We extend the Android middleware and the underlying Linux kernel to deny IPC among applications belonging to different domains. Moreover, TrustDroid enforces data filtering on default Android databases (e.g., *Contacts*, *SMS*, etc.) so that applications have access only to the data subset of the their respective domains.

Filtering Network Traffic: We modified the standard Android kernel firewall to enable network filtering and socket control. This allows us to isolate network traffic among domains and enables the deployment of basic context-based policies for the network traffic.

File System Control: We extend the current Android Linux kernel with TOMOYO Linux based mandatory access control and corresponding TOMOYO policies to enforce domain isolation at the file system level. This allows us to constrain the access to worldwide readable files to one specific domain. To the best of our knowledge, TOMOYO has never been applied on a real Android device (e.g., Nexus One) before.

Integration in Trusted Infrastructures: Our design includes essential properties and building blocks for integrating Android-based smartphones into sophisticated trusted infrastructures, such as Trusted Virtual Domains [38].

We have tested TrustDroid with Android Market applications and show that it induces only a negligible runtime overhead and minimally impacts the battery life-time.

3.4 Technical Problem Description and Model

We consider a corporate scenario which involves the following parties: (i) an enterprise (a company), (ii) a device (a smartphone), and (iii) an employee (the smartphone user). The enterprise issues mobile devices to its employees. The employees use their device for business related tasks, e.g., accessing the corporate network, loading and storing confidential documents, or organizing business contacts in an address book. To perform these tasks, the enterprise either deploys proprietary software on the device, e.g., a custom VPN client including the necessary authentication credentials, or provides a company-internal service, e.g., enterprise app market, from which employees can download and install those apps.

In this scenario, the enterprise is an additional stakeholder on the employees' devices and requires the protection of its deployed assets (software and data). Corporate assets may be compromised, e.g., when the user installs applications from untrusted public sources. Moreover, the employee accesses the enterprise internal network from his device and thus malware can potentially spread from the device into the corporate network.

A straightforward solution would be to prohibit any non-corporate app on the device (as proposed by, e.g., [37]). However, this is counter-intuitive to the idea of a *smartphone* and might even tempt employees to circumvent or disable this too restrictive security policy, e.g., by rooting the device. The default Android security mechanisms and recent extensions, on the other hand, are insufficient to provide enough isolation of untrusted applications and thus to protect the enterprise's assets. Virtualization can provide stronger isolation between trusted and untrusted domains (also its complexity can induce new security vulnerabilities [25, 26, 190, 208, 27, 28, 205]), but noticeably use up the battery life of the device, because major parts of the software stack are duplicated and executed in parallel in currently available virtualization solutions.

Consequently, an isolation solution is required, which preserves the battery life by minimizing the computational overhead and still provides isolation of corporate assets from untrusted applications.

3.4.1 Adversary and Trust Model

We consider *software* attacks launched by the adversary on the device at different layers of the Android software stack. The adversary's goal is to get access to corporate assets, e.g., to steal confidential data, to compromise corporate applications, or to infiltrate the corporate network. The adversary can penetrate the system by injecting malware (e.g., by spreading it through the Android Market) or by exploiting vulnerabilities of benign applications. Malicious applications may either be granted by the user the privileges to access sensitive resources (see Gemini [113]) or try to extend their privileges by launching privilege escalation attacks [30, 161, 111, 135, 136, 16].

We assume that the enterprise is trusted and that the employee is not malicious, i.e., he does not intentionally leak the assets stored on his device. However, he is prone to security-critical errors, such as installing malware. The device is generally untrusted, but has a Trusted Computing Base (TCB) that is responsible for security enforcement on the platform. The TCB is trusted by the enterprise.

3.4.2 Objectives and Requirements

We require the integrity and confidentiality of the corporate assets on the device, while preserving the usability. Furthermore, we require that the integrity of the corporate network will be preserved even if malware infiltrated employees' devices. With respect to these objectives, we define the following requirements:

Isolation. Corporate assets must be isolated in a separate domain from untrusted data and software, and any communication between different domains must be prevented. In particular, the following communication channels must be considered: IPC channels, the file system and socket connections. In addition, potential malware on the device must be prevented from accessing the corporate network.

Access control. Access by applications to assets stored on the device must be controlled by the enterprise through access control rules defined in a security policy, e.g., a new app can only be installed in the corporate domain if the policy states that it is trusted.

Legacy and Transparency. To preserve the smartphone's functionality, we require our solution to be compatible to the default Android OS and to 3^{rd} party applications. Further, it should be transparent to the employee.

Low overhead. With respect to the constrained resources of smartphones, in particular the battery-life, our solution has to be lightweight.

3.4.3 Assumptions

We consider the underlying Linux kernel and the Android middleware as Trusted Computing Base (TCB), and assume that they have not been maliciously designed. Moreover, we assume the availability of mechanisms on the platform to guarantee integrity of the TCB (i.e., OS and firmware) on the device. For instance, this can be achieved with secure boot which is a feature of off-the-shelf hardware (e.g., M-Shield [180] and ARM TustZone [5]) or software security extensions for embedded devices (e.g., a Mobile Trusted Module (MTM) [187]).

3.5 Design of TrustDroid

In this section we describe the design and architecture of TrustDroid. The main idea is to group applications into isolated domains. With isolation we mean that

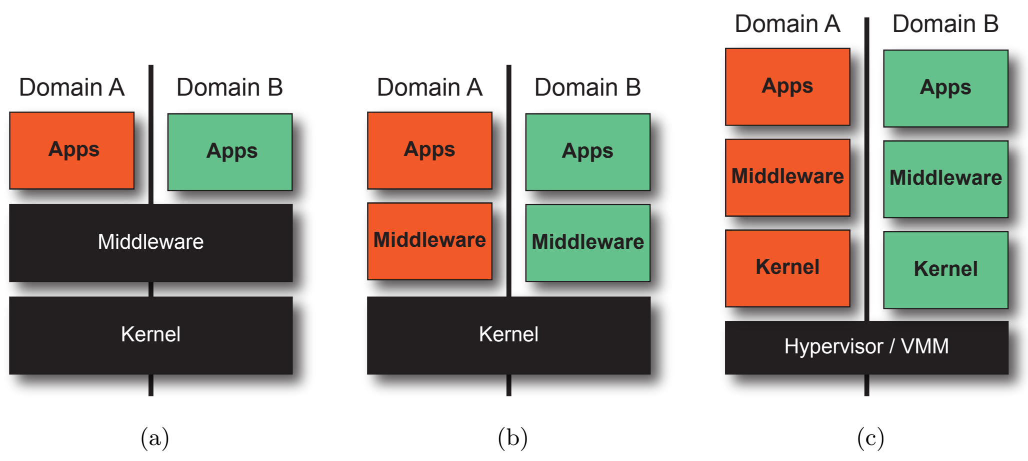

Figure 3.1: Approaches to isolation: (a) TrustDroid; (b) OS-level virtualization; (c) Hypervisor/VMM. Black blocks indicate the Trusted Computing Base of each approach.

applications in different domains are prevented from communicating with each other via ICC, Linux IPC, the file system, or a local network connection. Figure 3.1 illustrates different approaches to achieve isolation: (a) the approach taken by TrustDroid, which extends Android's middleware and kernel with mandatory access control; (b) OS-level virtualization, where each domain has its own middleware; (c) isolation enforced via a hypervisor and virtual machines, where each domain contains the full Android software stack. Comparing these approaches, TrustDroid has on the one hand the largest TCB, but on the other hand it is the most lightweight one, since it does not duplicate the Android software stack, and still provides good isolation, as we will argue in the remainder of this chapter.

Our extensions to the Android OS are presented in Figure 3.2. The middleware extensions consist of several components: *Policy Manager*, *Firewall Manager*, *Kernel MAC Manager*, an additional MAC for Inter Component Communication (ICC), and finally a modified *Package Manager*. The *Policy Manager* is responsible for determining the color for each installed application, for issuing the corresponding policies to enforce the isolation between different colors, and to enforce these policies on any kind of ICC. The *Firewall Manager* and the *Kernel MAC Manager* are instructed by the *Policy Manager* to apply the corresponding rules to enforce the isolation on the network layer and the kernel layer, respectively. To enforce the latter, TrustDroid relies on default features of the Linux kernel, which can also be activated in Android's Linux kernel: a firewall (FW) and a Kernel-level MAC mechanism. Since we modified the Android middleware, a company that wants to make use of TrustDroid has to roll out a customized version of Android to their employees' smartphones.

In the subsequent sections, we elaborate in more detail on the components of TrustDroid that enforce domain isolation.

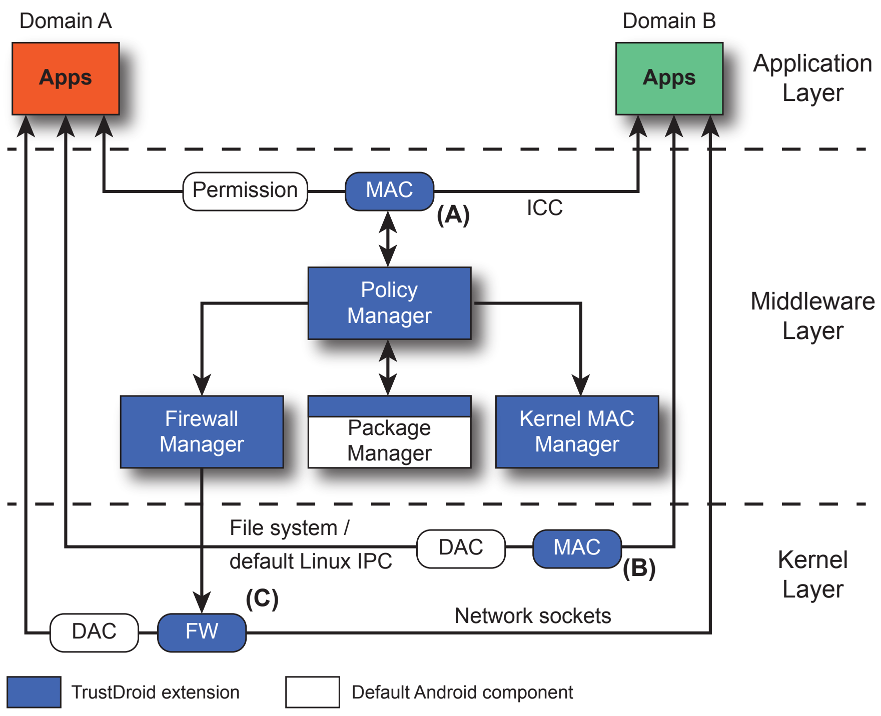

Figure 3.2: TRUSTDROID architecture with isolation of different colors (A) in the middleware, (B) at the file system/default Linux IPC level, and (C) at the network level.

3.5.1 Policy Manager

In this section we explain the *Policy Manager* component of TRUSTDROID and elaborate in more detail on how it colors applications and enforces domain isolation in the middleware.

3.5.1.1 Application Coloring

The fundamental step in our architecture to isolate apps is to assign each app a trust level, i.e., to *color* them. In TRUSTDROID, we assume three trust levels for applications: (1) pre-installed *system* apps, which include system *ContentProviders* and *Services*; (2) *trusted* third party apps provided by the enterprise; (3) *untrusted* third party apps, which are retrieved from public sources such as the Android Market. While trusted and untrusted apps must be isolated from each other, system applications usually have to be accessible by all installed applications in order to preserve correct functionality of those applications and sustain both transparency and legacy compliance of our solution (e.g., Android's application framework API).

In TRUSTDROID, system apps (i.e., pre-installed apps) are already colored during platform setup in accordance with the enterprise's security policies. Additionally installed third party apps are colored upon installation, before any code of the app is executed. In Android, the *Package Manager* is responsible for the installation of new applications and in TRUSTDROID we extended it to interface with the *Policy Manager*, such that the *Policy Manager* can determine the color of the new app, issue the necessary rules for its isolation in the middleware, and instruct the *Firewall Manager* and *Kernel MAC Manager* to enforce the corresponding policies on the lower levels.

Determining the color of an app can be based on various mechanisms. For instance, it can be based on a list of application hashes for each color or based on the information available about the new app, such as developer signature or requested permissions. For TrustDroid we opted for a certification based approach. The *Policy Manager* recognizes a special certificate (issued by the enterprise), which is optionally contained in the application package of apps. Based on this certificate, TrustDroid's *Policy Manager* verifies the authenticity and integrity of the new app. Moreover, the certificate may define a platform state, (e.g., the already installed applications), in which the certificate is valid only. A trusted service on the device is responsible for verifying these certificates. This service also measures the platform state, provides secure storage for the certificate verification keys, and maintains the verification key hierarchy such that only the enterprise can issue valid certificates. We use a *Mobile Trusted Module* (MTM) and as certificate format *Remote Integrity Metrics* (RIM) certificates, both defined by the Trusted Computing Group (TCG) [187]. We refer to Section 3.6 for more details on how we use and implement those.

If such a RIM certificate is present, it must be successfully verified to continue the installation, i.e., the certificate must have been issued by the enterprise, the application package's integrity must be verified, and the platform state defined in the certificate must be fulfilled. Otherwise, the installation is aborted. In case of a successful verification, the certificate determines the color of the new app. In our corporate scenario with only two application domains, successfully verified apps are in the *trusted* corporate domain. If no certificate is found, the app is by default colored as *untrusted*. This applies, for example, to all Android Market apps.

Alternatively, the certificates can already be pre-installed on the phone and the *Policy Manager* checks for a pre-installed certificate corresponding to the new app.

Generating the RIM certificates for applications requires a corresponding PKI inside the company. However, almost all companies today have integrated a PKI into their IT infrastructure that can easily serve for this purpose. For the initial setup of the mobile devices the certificates are generated and integrated once for every pre-installed trusted application. By integrating the deployment of RIM certificates into a mobile device management solution or a company internal app market the process of app-certification can be automated for updates or applications installed later.

3.5.1.2 Inter Component Communication

As described in our Background Chapter 2, Android uses Inter Component Communication as the primary method of communication between apps. Although ICC is technically based on IPC at the kernel level, it can be seen as a logical connection in the middleware. Thus, enforcement of isolation in the middleware has to be implemented based on access control on ICC. In general, one can distinguish different kinds of ICC that can be used by apps for communication.

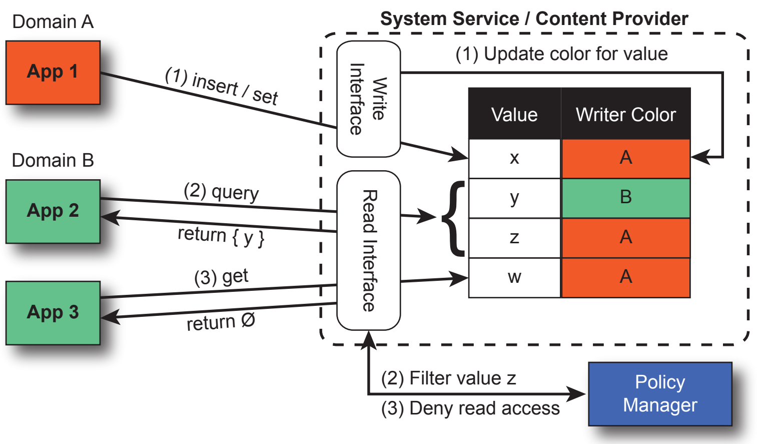

Figure 3.3: Coloring of data (1) and isolation of data from different colors in the (2) system Content Providers and (3) system Service.

Direct ICC. The most obvious way for apps to communicate via ICC is to establish direct communication links. For instance, an app could send an *Intent* to another app, connect to its *Service*, or query the *ContentProvider* of another app. The TrustDroid MAC on ICC detects this communication and prevents it in case the sender and receiver app of the ICC have different colors. It thereby acts as an additional MAC besides the default access control of Android (i.e., permission checks). As mentioned in Section 3.5.1, system apps form an exception and direct ICC is never prohibited if either sender or receiver of the ICC is a system app.

If two applications depend on each other, it is the responsibility of the certificate issuer, i.e, the enterprise in our scenario, to take care that these applications are in the same domain and to resolve any conflict in case the applications should have different trust levels according to the issuer's security policy.

Broadcast Intents. Besides the obvious direct ICC, apps are also able to send broadcast *Intents*, which are delivered to all registered *Receivers*. Similar to the approaches taken in [140] and [P5], TrustDroid filters out all *Receivers* of a broadcast that have a different color than the sender. Again, system apps are an exception and are not filtered from the *Receivers* list.

System Content Providers. A mechanism for apps from different domains to communicate indirectly is to share data in system *ContentProviders*, such as the *ContactsProvider* database, the *Clipboard*, or the *Calendar*. The ICC call to read data from such a provider does not give any information on the origin of the data, i.e., who wrote the data to the provider. We achieve domain isolation for system Providers as depicted in Figure 3.3. TrustDroid extends the system *ContentProviders* such that all data is *colored* with the color of its originator app (Step (1) in Figure 3.3). Upon read access to a provider, all data colored differently than the reader app is filtered from the response (Step (2) in Figure 3.3).

System Service Providers. A covert method for apps to communicate are system *Services*, such as the *AudioManager* [161]. However, in our adversary model (cf. Section 3.4.1), we assume that corporate apps are trusted and not malicious and thus no sender for such a covert channel exists in the trusted corporate domain. Nevertheless, data might leak via system *Services* from the trusted to the untrusted domain and thus isolation should be enforced here as well. Thus, as for the system *ContentProviders*, TrustDroid tags the read-/writable data values of the system *Services* with the color of the last app updating them, e.g., when setting the volume level (Step (1) in Figure 3.3). Read access to these values is denied in case the colors of the reader and the data differ (Step (3) in Figure 3.3). Although this approach does not prevent this kind of covert channel per se, it drastically reduces its bandwidth to 1-bit, because the reader only gains information if his corresponding writer changed the value or not.

Alternatively, TrustDroid could return a pseudo or null value instead of denying the read access. However, in contrast to system *ContentProviders*, on which a read operation by design might return an empty response, system *Services* are expected to return the requested value. Thus, returning a pseudo or null value may crash the calling app, or even cause more severe harm to the hardware or user, for instance, if the app reads a very low volume level when instead the real volume level is very high.

3.5.2 Kernel MAC Manager

The *Kernel MAC Manager* is responsible for communication with and management of the MAC mechanism provided by the underlying Linux kernel. Such mechanisms, like SELinux [114] or TOMOYO Linux [87], are already by default features of the Linux kernel and provide mandatory access control on various aspects of the OS, including the file system and the Inter-Process Communication. Thus, by employing such a MAC mechanism, TrustDroid achieves the isolation of domains on file system and IPC level. More explicitly, we create a MAC domain for each color and each app is added to the domain of its color upon installation. The *Policy Manager* instructs the *Kernel MAC Manager* to which domain a new application has to be added and the *Kernel MAC Manager* translates this instruction into low-level rules, which are inserted into the MAC mechanism and which define the isolation of domains at file system and IPC level.

File System. The file system is a further communication channel for applications. Apps are able to share files system-wide, by writing them to a system-wide readable location. Thus, a sending application can write such a file and a receiving application would simply read the same file. The mandatory access control mechanism enforces isolation on the file system in addition to the discretionary access control applied by default. TrustDroid applies rules, which enforce that a system-wide readable file can be only read by a another app of the same color as the writer. Thus, if an app declares a file system-wide readable, it is shared only within the domain of the writer.

Moreover, mandatory access control can, with corresponding policies, even be used to constrain the superuser (*root*) account. Hence, even if a malicious application gains superuser privileges, it's file system scope could be limited to it's domain.

Inter-Process Communication. To prevent any communication of apps through default Linux IPC (e.g., pipes, sockets, messages, or shared memory), TrustDroid leverages the same domains already established for the file system access control. Thus, apps are not able to establish IPC with differently colored apps. However, system applications form an exception, since denial of communication to system apps renders any application dysfunctional.

Potentially, ICC, which is based on Binder IPC, can be essentially addressed with kernel level MAC. However, in this case the policy enforcement would be limited to direct ICC between apps and would miss indirect communications, e.g., via *ContentProviders* or *Broadcast Intents*. In this sense, MAC on Binder is supplementary to the ICC MAC, because it enforces policies even in case (malicious) applications manage to disable the ICC MAC.

3.5.3 Firewall Manager

A further channel that has to be considered is Internet networking, i.e., network sockets used for communication via Internet protocols (such as TCP/IP). Based on these sockets applications are able to communicate with remote hosts, but also with other applications on the same platform. Thus, isolation with respect to the corporate smartphone scenario has to take both local and remote communications into consideration. To enforce isolation, TrustDroid employs a firewall to modify or block Internet socket based communication. Managing the firewall rules based on the policies from the *Policy Manager* is the responsibility of the *Firewall Manager* component.

Local Isolation. To locally enforce isolation between domains on the platform, TrustDroid prohibits any communication from a local network socket of an untrusted application to another local network socket via the loopback device. Although, on first glance, this might appear over-restrictive, it is a reasonable enforcement, because applications residing on the same platform usually employ lightweight ICC to communicate instead of network channels.

Alternatively, network communication within each domain could be allowed and only cross-domain traffic be prevented. However, this would require that the *Firewall Manager* knows which Internet socket belongs to which application and which address has been assigned to each socket.

Remote Isolation and Context-Awareness. Enforcing isolation between domains on the network traffic between the platform and remote hosts, e.g., web-servers, is a harder problem than local enforcement. All data that leaves the phone is beyond the policy enforcement capabilities of TrustDroid. For instance, applications in different domains could exchange data via a remote web-service. Moreover, with respect to the corporate scenario, one must consider that malware on the phone might spread into the corporate network once the phone connects to it.

To address the former problem, TrustDroid uses a firewall that is able to tag (*color*) the network traffic, e.g., VLAN. If the network infrastructure supports the isolation of traffic, for instance in Trusted Virtual Domains (TVDs) [38], the policy enforcement is extended beyond the mobile platform.

To address the latter problem, TrustDroid employs context-aware policy enforcement on outgoing traffic. The context can be composed of various factors, for instance, the absence/presence of a user, the temperature of the device, or the network state. In TrustDroid, the context means the physical location of the device and the network the device is connected to. Each context definition is associated with a policy that defines how to proceed with the network traffic of untrusted applications, e.g., blocking all traffic or manipulating it in a particular way. Thus, if the platform is physically on corporate premises or connected to the corporate network, all untrusted, non-corporate apps could be denied network access or their traffic can be manipulated, for instance, to reroute it to a security proxy or an isolated guest network.

3.6 Implementation and Evaluation

3.6.1 Implementation

We implemented TrustDroid based on the Android 2.2.1 sources and the Android Linux kernel version 2.6.32. We extended the default Android *ActivityManagerService* with a new component for the TrustDroid *Policy Manager* and the additional policy enforcement on ICC. We implemented the *Firewall Manager* and *Kernel MAC Manager* as new packages in the system *Services* in the middleware.

The *Policy Manager* contains a minimal native MTM implementation, which is loaded as a shared library and called via the Java Native Interface (JNI). Alternatively, TrustDroid could use more sophisticated and secure MTM implementations as proposed in [S8, 203, 213]. The MTM provides the means to verify Remote Integrity Metrics (RIM) certificates, to measure the software state of the platform, and to securely maintain monotonic counters.

Figure 3.4 illustrates the control flow for coloring a new application during installation and mapping the policies from the *Policy Manager* to the kernel and network level. Solid lines illustrate the control flow in case the application package contains a RIM certificate. Dashed lines show the alternate flow in case no RIM certificate is included in the package.

Application Coloring. To color new apps during installation, we extended the Android *Package Manager* to call the TrustDroid *Policy Manager* during the early installation procedure (step 1 in Figure 3.4) in order to verify the certificate potentially included in the application package (denoted APK) and determine the color of the new app. Therefore, the certificate is first extracted from the APK (steps 2 and 3a) and the resulting APK is verified with this certificate (steps 4a and 5a). In case the verification fails, the installation is aborted by throwing a Security Exception back to the *Package Manager* (step 6a). In case no RIM certificate is contained in the APK, the installation proceeds normally (step 3b). If the installation is continued and succeeds (step 7), a second remote call from the *Package Manager* informs the *Policy Manager* about this success (step 8) and thus triggers the issuing of corresponding policies to isolate the new app

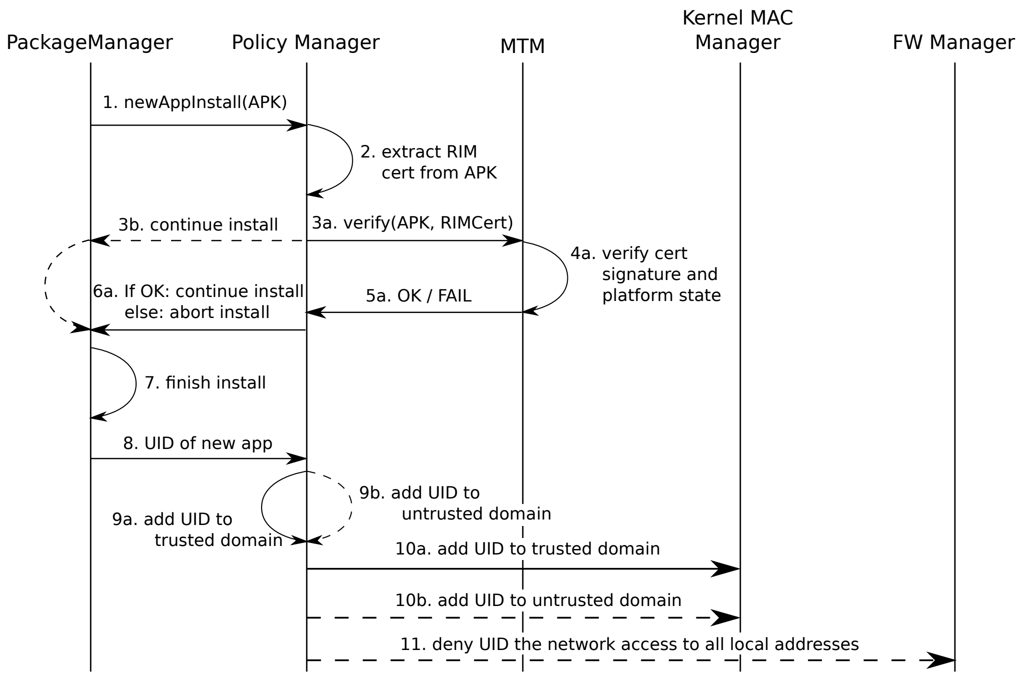
PackageManager
Policy Manager
MTM
Kernel MAC Manager
FW Manager
1. newAppInstall(APK)
2. extract RIM cert from APK
3b. continue install
3a. verify(APK, RIMCert)
4a. verify cert signature and platform state
6a. If OK: continue install else: abort install
5a. OK / FAIL
7. finish install
8. UID of new app
9b. add UID to untrusted domain
9a. add UID to trusted domain
10a. add UID to trusted domain
10b. add UID to untrusted domain
11. deny UID the network access to all local addresses

Figure 3.4: Control flow for the installation of a new application in case the installation package contains a RIM certificate (solid lines) or no RIM certificate is included in the package (dashed lines).

from other apps with a different color at ICC level (steps 9a and 9b), at file system and IPC level (steps 10a and 10b), and the network level (step 11). Only after this final step and after all policy changes have been issued, the *Package Manager* allows the new application to be started, thus preventing a potential race condition between putting isolating policies into effect and executing untrusted code.

RIM Certificates and Life-Cycle Management. As certificate format, we chose the RIM certificates as defined in the TCG Mobile Trusted Module (MTM) specifications [187]. In addition to the authenticity and integrity verification provided by other certificate standards such as X.509, RIM certificates additionally provide valuable features for a trusted life-cycle management. RIM certificates define a platform state in which the certificate is valid. This state is composed of monotonic counter values of the MTM and the measured software state. If either the counter value or software state defined in a RIM certificate mismatches the corresponding value of the MTM, the certificate verification fails. RIM certificates are signed with so-called *verification keys*. These verification keys form a key hierarchy, whose root key can be exclusively controlled by a particular entity, the enterprise in our scenario. Thus, only the enterprise is able to create valid RIM certificates for its employees' devices and, thus, only successfully certified apps are considered as trusted. Examples for MTM-based enhanced life-cycle management of apps are the prevention of version rollback attacks based on monotonic MTM counters, the binding of the installation to a certain platform state, or the trustworthy reporting of the software state, i.e., installed applications. To certify APKs, we developed a small tool written in Java and that makes use of the jTSS[1].

Network, Default IPC, and File System Isolation. To implement isolation at network level, default Linux IPC, and file system level, our implementation employs *netfilter*[2] that is by default present in the kernel as well *TOMOYO Linux*[3] v1.8 that is available as a kernel patch. To maintain them from the *Firewall Manager* and *Kernel MAC Manager*, respectively, we cross-compiled and adapted the user-space tools *iptables* and *ccs-tools*. The former is used to administrate netfilter and the latter for TOMOYO Linux policy management.

In TrustDroid, we created two TOMOYO Linux domains for third party apps, *trusted* and *untrusted*, and policies that isolate these domains on file system and default Linux IPC level. Upon installation of a new application, the UID of the new application is inserted into either one of those domains (steps 10a and 10b in Figure 3.4). A third domain for system apps is accessible by both the trusted and untrusted domains.

By default, TrustDroid denies all untrusted applications network communication to local addresses on the phone and the *Policy Manager* instructs the *Firewall Manager* to enforce this isolation also for newly installed untrusted applications (step 11 in Figure 3.4). Thus, any local network communication between trusted apps is isolated from untrusted apps.

A particular technical challenge was the adaption of the TOMOYO Linux user-space programs, in order to be able to maintain the TOMOYO Linux policies locally on the device. Recent documentation for TOMOYO Linux on the Android emulator describes the policy administration from a remote host instead of locally on the device and thus required certain adaption for TrustDroid.

Although TOMOYO Linux in version 1.8 provides MAC for Internet sockets as well, thus the means to exclude applications with fine-grained policies (e.g., UID, port or IP address) from Internet access, we opted for netfilter for two major reasons: (1) Unexpectedly denying access to sockets is much more likely to crash affected applications, in contrast to simply blocking the outgoing traffic and thus faking a disabled network connection; (2) netfilter provides much more flexibility than simply access control, e.g., manipulating or tagging network traffic for advanced security infrastructures such as TVDs or security proxies.

An alternative building block to TOMOYO Linux would be SELinux, which is by now a default component of the Android codebase since version 4.3. SELinux is based on extended file attributes and thus provides a more intuitive solution for domain isolation at the file system level. On the other hand, it is more complex to administer than TOMOYO Linux and requires modifications to the default Android file system, because the default file system at the time this work was conducted did not support extended file attributes. In addition, the SE Android [175] project has since identified further technical challenges in adapting SELinux for Android.

[1]`http://trustedjava.sourceforge.net/`
[2]`http://www.netfilter.org/`
[3]`http://tomoyo.sourceforge.jp/`

Context-Awareness. A context in our current implementation is simply the definition of a WiFi state and/or location. For instance, it could be the SSID of the wireless network, the MAC address of the access point, a certain latitude/longitude range, or proximity to a certain location.

To implement the context-aware management of the netfilter rules (cf. Section 3.5.3), the current *Firewall Manager* uses two state listeners—one for changes of the WiFi state and one for updates on the location. The former is simply a *Receiver* for notification *Broadcasts* about the changed Wifi state. The latter is an *LocationListener* thread registered at the *LocationManager*. In case one of the two listeners is triggered, the new state is compared with the installed contexts and the policies of any matched context are activated. The active policies of contexts that are not fulfilled anymore are revoked.

Middleware Isolation. The implementation of the additional policy enforcement on ICC is based on the XManDroid framework presented in our technical report [T4], which provides the necessary hooks in the Android middleware to easily implement policy enforcement on direct ICC, *Broadcast Intents*, and channels via system *ContentProviders* and *Services*.

To prevent direct ICC between applications with different colors, we wrapped the *checkPermission* function of the *ActivityManagerService*, which is called every time a new ICC channel should be established. If the default MAC of Android permits the new ICC, TrustDroid performs an additional check to compare the colors of the caller and callee. On mismatch, the previous decision is overruled and the ICC denied. To prevent data flows between different domains via *Broadcast Intents*, our implementation is similar to [140] and implements hooks in the broadcast management in the *ActivityManagerService* to filter out all receivers of a *Broadcast* that do not have the same color as the sender. As described in Section 3.5.1.2, we extended the interfaces of system *ContentProviders*, such as *Settings* or *Contacts*, and of the system *Services*, such as the *AudioManager*, to color data upon write access and filter data/deny access upon read access.

Moreover, the *PackageManagerService* allows applications to iterate over the information of installed packages, e.g., to find a specific application that might provide supplementary services. In TrustDroid we extended this functionality with additional filters, such that applications can only receive a list of information about applications of the same color or about system applications.

Prototype Policy. In our prototypical implementation we use a strict domain isolation of the form

$$Subject_{Color} == Object_{Color}$$

meaning that access to an object by a subject is only granted if the subject and object are in an identical domain. Only exceptions from this rule are the system domain applications, to which access from other domains must be granted in order to preserve operability and legacy-compliance of 3^{rd} party apps, although each system application internally enforces access control on data they manage in order to prevent cross-domain information flows (see Section 3.5.1.2).

Although more advanced and fine-grained policies might be desirable (e.g., allowing uni-directional information flows from one domain to another) and are also technical feasible (for instance, by explicitly considering the domain of the subject), this stricter policy of our prototype forms the foundation on top of which any kind of (advanced) domain isolation in the TrustDroid scenario can be built through refinement and defining exceptions. Moreover, this stricter policy is simple enough to avoid the need for a dedicated policy language at this prototypical state and allowed us to hardcode the policy into the policy enforcement points in the middleware and in the TOMOYO Linux policy.

3.6.2 Evaluation

We evaluated the performance overhead and memory footprint of our extensions to the middleware with 50 apps from the Android Market, categorized in two domains (plus one domain for system apps). On average our additional policy enforcement on ICC added 0.170 ms to the decision process on whether or not an ICC is allowed (default Android requires on average 0.184 ms). The standard deviation in this case was 1.910 ms, caused by high system-load due to heavy multi-threading during some measurements. The verification of the RIM certificate during the installation of new packages required on average 869.750 ms with a standard deviation of 645.313 ms. The average memory footprint of our extensions to the Android system server was 348.2 KB with a standard deviation of 200.8 KB, which is comparatively small to the default footprint of approximately 2 MB.

In our prototype implementation, the policy file of TOMOYO consumes on average a little more than 200 KB of memory. The policy file includes access control rules for file system and standard IPC mechanisms e.g. communication based on Unix domain sockets.

3.7 Discussion

In this section we discuss the security of TrustDroid and highlight possible extensions.

3.7.1 Security Considerations

The main goal of TrustDroid is to provide an efficient and practical means to enforce domain isolation on Android. In particular, TrustDroid isolates applications by their respective trust levels, meaning that applications have no means to communicate with each other if their trust levels mismatch. Our requirement of access control is achieved by including certificates into an application package. Further, to control as many communication channels as possible, TrustDroid targets different layers of the Android software stack. First, IPC traffic (in the middleware and the kernel) is completely mediated by TrustDroid and is target to domain policies. Hence, malicious applications cannot use interfaces of applications belonging to other domains, even if the interfaces are exposed as public. Thereby TrustDroid prevents privilege escalation attacks from affecting other domains. Second, TrustDroid prevents unauthorized data access, by performing fine-grained data filtering on application data and data stored in common databases (*SMS*, *Contacts*, etc.). In particular, this prevents malicious applications from reading data of the corporate domain, as long as the malicious application has not been issued by the enterprise itself, which is excluded in our adversary model (cf. Section 3.4.1). Third, TrustDroid successfully mitigates the impact of root exploits, because our TOMOYO policies overrule the Linux discretionary access control and prevent an adversary from accessing files of another domain. Finally, communication over socket connections are constrained to the domain boundary.

Although our approach is lightweight and practical, it does not provide the same degree of isolation as full-virtualization would do. In particular, TrustDroid only mitigates kernel-level attacks by restricting access to the file system, but in general, it cannot prevent an adversary from compromising the Trusted Computing Base (TCB), which for TrustDroid includes the underlying Linux kernel and the Android middleware (see Section 3.4.3). In practice, static integrity of the TCB can be ensured by means of secure boot [212]. However, the TCB is still vulnerable to runtime attacks subsequent to a secure boot. Solving this problem is orthogonal to the solution presented in this work and recently different solutions for ensuring kernel integrity on mobile devices have been brought forward [6, 63].

The primary cause for runtime attacks on Android is the deployment of native code (shared C/C++ libraries) [136]. Although Android applications are written in Java, a type-safe language, the application developers may also include (custom) native libraries via the Java Native Interface (JNI). Moreover, many native system libraries are mapped by default to the program memory space. A straightforward countermeasure against native code attacks would be to prohibit the installation of applications that include native code. However, this is rather restrictive and, similar to prohibiting any non-corporate app (cf. Section 3.4), contradicts the actual purpose of smartphones or might even tempt the phone user to break the security mechanisms in place. Another approach to address native code attacks is Native Client [164], which provides an isolated sandbox for native code. However, this solution requires the recompilation of all available applications that contain native code—an infeasible assumption with respect to legacy compliance given the huge number of apps in the Android Market.

Moreover, as argued and shown in [212], mandatory access control can also be efficiently deployed on mobile platforms to enforce isolation for the complete Linux kernel. We consider this as a valuable extension to TrustDroid to mitigate kernel attacks, which could easily be integrated in TrustDroid, since a kernel-level MAC mechanism is already a building block of our design (see Section 3.5).

Finally, TrustDroid uses a separate, accessible domain for system applications and services, which is due to the fact that all applications require these system apps to work correctly. If an adversary identifies a vulnerability in one of these applications, he may potentially circumvent domain isolation and access data not belonging to his domain. However, until today, vulnerabilities of system applications were constrained to confused deputy attacks and did not allow an adversary to access sensitive data [148]. Protecting system applications and services from being exploited is orthogonal to harden the kernel, and we aim to consider this in our future work. Alternatively, one could deploy apps in the trusted domain, which offer the functionality of certain system apps (e.g., business contacts app or enterprise browser) and isolate the now redundant system apps by classifying them as untrusted. In fact, recent extensions to Android (after this work was conducted) in context of Google's Android for Work initiative (or Samsung's Knox) followed exactly this design pattern.

3.7.2 Trusted Computing

Our TrustDroid design leans towards possible extensions with Trusted Computing functionality.

Currently, we leverage a Mobile Trusted Module (MTM) to validate application installation packages and to determine their color. The features of the employed RIM certificates in contrast to established certification standards such as X.509 provide the means for an enhanced life-cycle management based on monotonic counters and the platform state, e.g., version rollback prevention. The current implementation of our MTM is simple, but more sophisticated approaches may be integrated into our current design [S8, 203, 213].

Moreover, our design includes the foundation for the integration of Trusted Computing Group (TCG) mechanisms such as the attestation of the domains [134, 99, 128], e.g., in the context of Trusted Network Connect [188] or remote data access [157], or the isolation of network traffic for infrastructures like Trusted Virtual Domains [38].

3.8 Related work

In this section we provide an overview of related work for establishing security domains and enforcing security policies on Android. We first discuss in Section 3.8.1 the related work at the time this work was conducted in order to underline the novelty of our results. We then discuss in Section 3.8.2 further related work that has been published after our work had been presented.

3.8.1 Status Quo at Time of Publication

3.8.1.1 Virtualization

A "classical" approach from the desktop/server area to establish isolated domains on the same platform is based on virtualization technologies. This approach has been ported to the mobile area [11, 165]. Although virtualization provides stronger[4] isolation, it duplicates the entire Android software stack, which renders those approaches quite heavyweight in consideration of the scarce battery life and memory capacity of smartphones. Possible approaches to mitigate this problem could be the automatic hibernation of VMs currently not displayed to the user, or the design of a *just-enough-OS/Middleware* to minimize the resident memory footprint of domains. However, at this time available mobile virtualization technology did not provide these features. In contrast, our solution is more lightweight, since the creation of a new domain simply requires the definition of a new string value (*color*) and deployment of a new MTM verification key. Moreover, from our past experience with mobile virtualization technology [31], we conclude that

[4]Although virtualization has been in the past not without its own security problems [25, 26, 190, 208, 27, 28, 205] due to its large management layer.

our solution is more practical in the sense that it is more portable to new hardware, because we can re-use the provided proprietary hardware drivers, while virtualization requires new (re-implemented) drivers or an additional *driver-domain* that multiplexes the hardware between the VMs (e.g., *dom0* in Xen [96]).

3.8.1.2 Kernel-level Mandatory Access Control

Another well established mechanism, that is at this time being ported to the Android platform, is kernel-level mandatory access control like SELinux or TOMOYO [166, 29]. These mechanisms allow, e.g., policy enforcement on processes, the file system, sockets, or IPC. In *SEIP* [212], SELinux was used to establish trusted and untrusted domains on the Linux Mobile (LiMo) platform in order to protect the platform integrity against malicious third party software. The work further shows how unique features of mobile devices can be leveraged to identify the borderline between trusted/untrusted domains and to simplify the policy specification, while maintaining a high level of platform integrity. The authors of [152] show how policies in the context of multiple mobile platform stakeholders can be created dynamically and present a prototype based on SELinux. Low-level mandatory access control is an essential building block in our design (see Section 3.5). However, it is insufficient for isolating domains on Android, because it does not consider the Android middleware system components, such as system *ContentProviders*/*Services* or *Broadcast Intents*, as communication channels between domains (see Section 3.4). Without high-level policy enforcement in the middleware, low-level MAC mechanisms can only grant/deny applications the access to system *ContentProviders* and *Services* as a whole. However, generally denying an app access to system components most likely crashes this app or at least renders it dysfunctional. Moreover, although these mechanisms allow to some extend fine-grained access control policies on the network, they do not support the manipulation of network packets like netfilter does (cf. Section 3.5.3). Nevertheless, the approach of *SEIP* [212] could enhance the integrity protection of our TCB (see Section 3.7).

3.8.1.3 Android Security Extensions

In the last two years, a number of security extensions to the Android security mechanisms have been introduced [24, 133, 139, 44, 46, P5]. Based on very similar incentives to TrustDroid, *Porscha* [139] proposes a DRM mechanism to enforce access control on specifically tagged data, such as SMS. However, this approach is limited to isolate data assets, but is not suitable to isolate particular (sets of) apps.

Similarly, the *TaintDroid* framework [46] tracks the propagation of tainted data from sensible sources (in program variables, files, and IPC) on the phone and detects unauthorized leakage of this data. However, it is limited to tracking data flows and does not consider control flows. Moreover, it does not enforce policies to prevent illegal data flows, but notifies the user in case an illegal flow was discovered. Nevertheless, TaintDroid could form a very valuable building block in our TrustDroid design to isolate data assets, if it would be extended with policy enforcement.

Both *APEX* [133] and *CRePE* [24] focus on enabling and disabling functionality and enforcing runtime constraints. While APEX provides the user with the means to selectively choose the permissions and runtime constraints each application has (e.g., limited number of text messages per day), CRePE enables the enforcement of context-related policies of the user or a third party (e.g., disabling bluetooth discovery). In this sense, both are related to our design goal to isolate untrusted applications based on the context (cf. Section 3.5.3) or protect data assets in shared resources like system *ContentProviders*. However, the enforcement described in APEX [133] and CRePE [24] is too coarse-grained. For instance, networking would be disabled for all applications, not just selected ones, or not only the access to certain data but to the entire *ContentProvider* would be denied to selected applications.

Saint [140] introduces a fine-grained, context-aware access control model to enable developers to install policies to protect the interfaces of their apps. Although Saint could, with a corresponding system centric policy, provide the isolation of apps on direct and broadcast ICC, it can not prevent indirect communication via system Components (see Section 3.5.1.2). However, Saint's design provided valuable input for the placement of authorization hooks in our TrustDroid design.

Our XManDroid [T4] framework addresses the problem of ICC-based privilege escalation by colluding apps and is also able to enforce policies on ICC channels via system components. The XManDroid framework as presented in our technical report [T4] formed the basis for our TrustDroid implementation, but had to be extended to enable application coloring and mapping of policies for domain isolation from the middleware onto the network and kernel level.

In general, none of these extensions provides any policy enforcement on the file system, IPC, or local Internet socket connections in order to enforce isolation of domains. However, TaintDroid with its data flow tracking mechanism has the potential to implement fine-grained policy enforcement.

3.8.2 Related Work Post-Publication

Since we presented our TrustDroid, a number of related work has been brought forward that follows very similar incentives or that realizes some of the ideas we discussed in our original related work section.

The increasing popularity of using Android-based smart devices (smartphones and tablets) in business contexts and the associated demand for better support of *bring-your-own-device* (BYOD) deployments, has prompted Google to initiate its *Android for Work* [80] program. With Android for Work, the platform support for securely deploying and managing corporate assets on the end-device has greatly increased. Most notably, Google incorporated Samsung's *Knox* [116] security framework, which was specifically designed to isolate private from business data/apps. Like TrustDroid, Knox (and now vanilla Android) build on mandatory access control instead of virtualization to establish isolation and even apply very similar design patterns to TrustDroid (although naturally in a more conservative manner).

Simultaneously to TrustDroid, the *Cells* [4] and *L4Android* [108] virtualization solutions have been proposed. Cells implements a lightweight OS-level virtualization based on Linux containers, which allows multiple virtual smartphones to execute simultaneously. In this sense, Cells resembles the approach 3.1(b) in Figure 3.1 (on page 22). To address the problem of code and data duplication that motivated our TrustDroid approach, Cells uses a novel file system layout based on unioning, which increases the sharing of common read-only code and data between the virtual phones. L4Android, in contrast, encapsulates the Android OS in a virtual machine that executes on top of the L4 microkernel and compartmentalizes basic services as native processes of the microkernel. The authors of L4Android also envision one dedicated VM for private and corporate purposes each. Thus, L4Android follows in principle the approach 3.1(c) in Figure 3.1. Although L4Android provides a hardware abstraction layer, it necessitates that drivers for each individual hardware platform are provided (or re-implemented) for their microkernel—a notorious problem for microkernel solutions unless they are supported by hardware vendors.

MOSES [156] also separates private from business data, but builds on top of the TaintDroid [46] framework pretty much in the spirit we discussed earlier for our TrustDroid. It defines different security profiles on a single device, which determine the access rights of applications executing under this profile. In addition to several authorization hooks in the Android middleware to isolate those security profiles, MOSES uses TaintDroid to taint data with its originating security profile ID and augments TaintDroid by enforcing access control policies based on the taints. Moreover, instead of relying on Kernel MAC like TrustDroid does, MOSES instruments several core libraries and components (such as the Binder library or Java Socket and OSFileSystem class) to intercept an app's interaction with the underlying Linux operating system.

AirBag [207] introduces a lightweight OS-level virtualization that creates an isolated runtime environment for apps with its own dedicated namespace and virtualized system resources using, for instance, Linux *cgroups*. Thus, in the spirit of fault isolation, malicious operations of isolated apps inflict damage only onto their isolated environment. Although primarily intended for sandboxing potential malware, AirBag's technique could be applied to isolate (untrusted) private apps on a corporate device.

PINPOINT [153] forgoes, in contrast to other solutions [207, 108, 4], a general-purpose resource isolation and instead introduces namespaces akin to Linux namespaces into the service management of Android's application framework to isolate very specific resources while allowing all other resources to function fully as intended. Thus, every application receives from the application framework, depending on the security policy, either a reference to the original service (e.g., location or telephony) or to a virtualized service, which filters the original data or returns mock data. Using this approach, apps in TrustDroid could be redirected to private or corporate services depending on the application color.

AppFork [138], like TrustDroid, extends the Android application framework to introduce different profiles that are isolated from each other. However, AppFork, in contrast to TrustDroid, allows the user to switch the profile of apps, while in TrustDroid each installed app is immutably assigned to one security domain. It uses different techniques, such as dynamic storage partitioning, to prevent cross-profile leaks during profile switching. Thus, AppFork comes technically close the Android multi-user accounts [70], where apps can be executed under different user profiles.

DeepDroid [195] enforces enterprise policies through dynamic memory instrumentation of critical system processes, such as Zygote or the system server, in the spirit of inlining reference monitors [48]. Thus, it avoids (in contrast to solutions like TrustDroid) the need to modify the firmware.

Aquifer [131] is a policy framework and enforcement system to mitigate accidental information disclosure (e.g., application-specific, user data objects such as officce documents, voice or written notes, and images). It is based on dynamically constructing the user interface workflow and enforcing export and app-involvement restrictions based on the runtime context and the purpose of the app (e.g., data intermediaries for *"share with"* functionality). Aquifer is primarily written to enforce restrictions along workflows and across applications and as such could also ensure that shared data does not cross a domain boundary. However, it is not designed for a general domain isolation.

As mentioned in our security discussion in Section 3.7.1, different ideas exist to further protect the Trusted Computing Base of TrustDroid. In recent years, orthogonal related work has introduced concrete solutions for those ideas. For instance, *NativeGuard* [181] forms a solution to decompose Android apps by extracting their native code libraries into separate service applications and hence confine native code to stricter policies. *Hypervision* [6] and *Sprobes* [63] both introduce novel techniques to protect the Linux kernel on mobile platforms by leveraging the TrustZone secure execution environment of modern ARM processors.

3.9 Conclusion

Google's Android did in its early version provide *no data and application isolation* between domains of different trust levels. In particular, there existed no efficient solution to isolate corporate and private applications and data on Android: the existing security extensions for Android only focus on one specific layer of the Android software stack, and hence, do not provide a general and system-wide solution for isolation.

In this work we presented TrustDroid, the first Android security framework that provided *practical and lightweight domain isolation* on Android, i.e., it does not affect the battery life-time significantly, requires no duplication of Android's software stack, and supports a large number of domains. In contrast to existing security extensions, TrustDroid enforces isolation on many abstraction layers: (1) in the middleware and kernel layer to constrain IPC traffic to a single domain, and to enforce data filtering for common databases such as Contacts, (2) at the kernel layer by enforcing mandatory access control on the file system, and (3) at the network layer to regulate network traffic, e.g., denying Internet access by untrusted applications while the employee is connected to the corporate network. Our evaluation results demonstrate that our solution adds a negligible runtime overhead, and in contrast to contemporary virtualization-based approaches [11, 108], only minimally affects the battery's life-time.

We also provided a detailed discussion on the design of TrustDroid and argue that TrustDroid can be used as a foundation for Trusted Computing enhanced concepts such as Trusted Virtual Domains (TVD), a distributed isolation concept known from the desktop world. In our future work, we aim to adopt domain isolation on the underlying Linux kernel so that an adversary can no longer exploit kernel vulnerabilities to circumvent domain isolation.

4 XManDroid

Towards Taming Privilege-Escalation Attacks on Android

4.1 Motivation

Android uses a permission-based authorization system to protect the user's privacy and the system's integrity against nosy or malicious apps (see Background Chapter 2). Before Google introduced app-vetting features into its app market (Google *Play*) and introduced an associated app verification service [73], this system relied solely on the user to consider and approve the permissions each app requests at install time. However, attacks demonstrated that even a vigilant end-user can be easily the victim of different forms of *application-level privilege escalation attacks* [30, 44, 148, 161] that allow a malicious app to violate the user's privacy or undermine the system integrity. Being able to mitigate such application-level privilege escalation attacks requires an Android security architecture that is able to detect the different forms of attacks and take appropriate counter-measures. Unfortunately, Android's original security architecture lacks the necessary authorization hooks and policy decision logic, and hence does not fulfill this requirement.

4.2 Problem Description

Confused Deputy and Collusion Attacks. Application-level privilege escalation attacks as in the context of this work are not based on classical software exploitation techniques or compromising the integrity of security critical system services. Instead, they exploit the fact that Android's security architecture was designed to constrain single application sandboxes (i.e., a single app or multiple apps with a shared UID), but is not capable of preventing permissions from being (mis-)used transitively across distinct apps. Two particular attack techniques, which are the focus of this work, are *confused deputy attacks* [30, 44, 148] and *collusion attacks* [161, 117].

In a *confused deputy attack* [88], a malicious but unprivileged app tricks a privileged app into misusing its permissions on behalf of the seemingly innocuous, malicious app. On Android, this vulnerability is often rooted in unprotected (or insufficiently protected) public interfaces of privileged apps, which thus turn into exploitable deputies. Confused deputy vulnerabilities have been shown by prior research to be present in both 3^{rd} party applications [30, 148] and Android system applications [44, 148]. For instance, on early versions of Android, an unprivileged app could send a specifically crafted *Intent* to the dialer application to trigger an unauthorized phone call [44].

On the other hand, in a *collusion attack*, the attacker splits the malicious functionality across two or more apps, each requesting only the permission(s) it requires for its particular functionality. Thus, each malicious application seems innocuous by itself, however, together they achieve a permission set that was not approved by the end-user. Collusion attacks require the malicious applications to communicate with each other. This communication can be either conducted via overt communication channels [117] (like ICC, file system, or shared components) or via covert communication channels [161] (like file locks or system services). Since such channels can be established logically at the middleware layer (e.g., ICC) or via kernel-managed resources (e.g. files and sockets), protection mechanisms must be deployed and synchronized at both the middleware and the kernel layer.

Previous Security Extensions to Android. Prior to this work, different solutions [44, 46, 36, 140, 148] targeted the problem of application-level privilege escalation attacks. However, none of those solutions considered satisfactorily both confused deputy and collusion attacks at the same time. The existing approaches require app developer participation to enforce policies or they enforce only very static policies that are too coarse-grained to effectively cover the attack vectors considered in this work. An important insight of our work is, that tackling both attack scenarios requires, in contrast to prior work, an approach that combines a system-centric solution with a flexible policy-driven enforcement while preserving system performance and compatibility to existing applications.

4.3 Our Goal and Contributions

In this work we address the problem of devising and building a security framework for Android that can mitigate confused deputy and collusion attacks. In contrast to prior work that aimed only at specific subclasses of those attacks, we aim for a general framework that allows a policy-driven protection against various variations of those application-level privilege escalation attacks. To support our design decisions, we conducted a heuristic analysis of the application communication patterns at runtime, based on a set of selected popular apps from the app market. Using these analysis results, attack patterns can be better distinguished from normal app operations.

The concrete contributions of this work are:

XManDroid *Security Framework:* We present the design and implementation of a security framework (dubbed XManDroid) to detect and prevent confused deputy and collusion attacks. To this end, we extend the application framework and kernel of Android's software stack: (1) new authorization hooks for *runtime* monitoring of Inter-Component Communication (ICC) between applications and of indirect communication through Android system components, such as system *Services* and *ContentProviders*. Inspired by the approach of *QUIRE* [36], we establish a semantic link between ICC calls that are checked at runtime by our authorization hooks in order to identify and prevent call-chains that form attack states; (2) we adapted TOMOYO Linux [87] as kernel-level Mandatory Access Control (MAC) to monitor and control the file system (i.e., files, Unix domain sockets, Internet sockets) access by application processes and thus supplement our middleware access control; (3) a runtime interaction between our security extensions to the Android middleware and to the kernel-level MAC. Although TOMOYO can intercept system calls and enforce MAC at the kernel-level, it lacks the high-level semantics and contextual information of the application framework to effectively enforce our security policies (e.g., based on the permissions an applications holds). To bridge this semantic gap between policy enforcement at the middleware layer and at kernel-level, we dynamically map the policies of the middleware to TOMOYO's policies.

Performance and effectiveness: Our prototypical implementation has a negligible performance overhead that is not noticeable by the user. We evaluated our implementation on a *Nexus One* development phone with 50 popular applications from the Android

Market and 25 test users (students). We also successfully evaluated our framework against application-level privilege escalation attacks presented in [44, 30, 148, 161]. In contrast to existing solutions (at the time of writing), our implementation detects all of those attacks including the sophisticated Soundcomber trojan [161] and an attack launched through a loopback device connection [30]. Finally, we discuss and evaluate possible problems, such as the rate of attack detection and falsely denied inter-app communication.

4.4 Requirements Analysis and Assumptions

In the following, we define our adversary model, devise our security requirements, and explain our assumptions.

4.4.1 Adversary Model

The goal of the attacker is violate the end-user's privacy by stealing user-private information from the device or to manipulate the system in ways that would require a higher privilege level than 3^{rd} apps usually have. To this end, the attacker does *not* aim at mounting software exploits that compromise other processes or at designing overly privileged apps that can single-handedly steal all user-private information. Instead the attacker aims at making his apps as seemingly innocuous as possible as to not trigger a vigilant user that inspects the apps' permissions.

We consider an attacker that is able to mount software-based attacks using one or more attacker apps that are installed on the end-user's device. Using these apps, the attacker is able to exploit confused deputies or perform collusion attacks. The attacker apps can interact with each other and with the system via the default application framework API (e.g., *Intent* messages or shared system components like *Contacts*) and through the default Linux IPC facilities (e.g., file system access, sockets, etc.). Thus, the attacker apps can use different communication channels like *direct communication* using ICC or sockets, *indirect but overt communication* via shared system components or the file system, or *covert communication channels* through shared system services or the file system. Moreover, the attacker is in full control of the application code of his apps and thus interaction with the system and other apps might occur from DEX bytecode as well as native code within the attacker's apps' sandboxes. This differs from previous work on application-level privilege escalation attacks [140, 36, 148], which aimed only at mitigating (unknown) confused deputy attacks over ICC, while we work towards an Android security framework that can address all above mentioned channels.

4.4.2 Objectives and Requirements

Ideally, a security solution to the confused deputy and collusion attack problem should be able to address all attacks that occur over the above mentioned communication channels between apps and the system. Thus, hardcoding specific enforcement policies is not an option. Instead, a more flexible and configurable security framework is required that supports security policies of different granularities depending on the concrete security demands at hand and that hence can target all or only a subset of the above mentioned communication channels. Moreover, a *system-centric* solution is required that does not rely on app developer participation in the enforcement and hence excludes a largely unreliable (or even malicious) actor in the app ecosystem. This requirement also entails the need for *legacy compatibility* to existing apps, since recompilation or reconfiguration of all existing apps is virtually impossible given the enormous number of apps on the market. Lastly, like any other security solution, in particular in the domain of resource constrained mobile devices, the security solution should only impose a *low performance overhead*.

4.4.3 Assumptions

We assume that the Trusted Computing Base (TCB), consisting of the Android middleware, application framework, and the underlying Linux kernel, is trusted and not malicious. However, pre-installed system applications may suffer from confused deputy vulnerabilities, as shown by different previous research works [44, 36, 148]. Further, we assume that the application framework and pre-installed applications are not free of covert channels, i.e., functionality that can be used to establish communication channels, as shown by previous work [161].

4.5 Framework Architecture

We first describe the basic idea behind the design of our security framework (Section 4.5.1) and explain then in detail the components of our framework and their interaction (Section 4.5.2). In this context, we also briefly introduce how our framework makes access control decisions based on a graph-based system representation and policy language.

4.5.1 Overview

The basic idea for our security framework is to perform runtime monitoring of the communication links between applications and to prevent communication links that can potentially form a collusion or confused deputy attack. Our approach is based on adding new reference monitors to the various middleware system components and the kernel that are critical for inter-application communication. These new monitors serve a double purpose: First, they enforce the security policies that prevent malicious communication links from being established. Second, they provide feedback to a central system view component of our framework, which represents the current system-wide inter-application communication state. This system view is the foundation for our graph-based security policies, which make use of the history of established (i.e., granted) communication links to derive whether a new communication link will lead to an attack state and hence has to be prevented. These policy decisions are made by a central policy decision point. Thus, our design follows the established design pattern of earlier access control systems, e.g., Flask [179], which discern policy enforcement logic from policy decision logic.

Our current reference monitors control communication channels that are located at the middleware (i.e., ICC and shared system components) as well as at the kernel-level (i.e., file system objects like files and sockets).

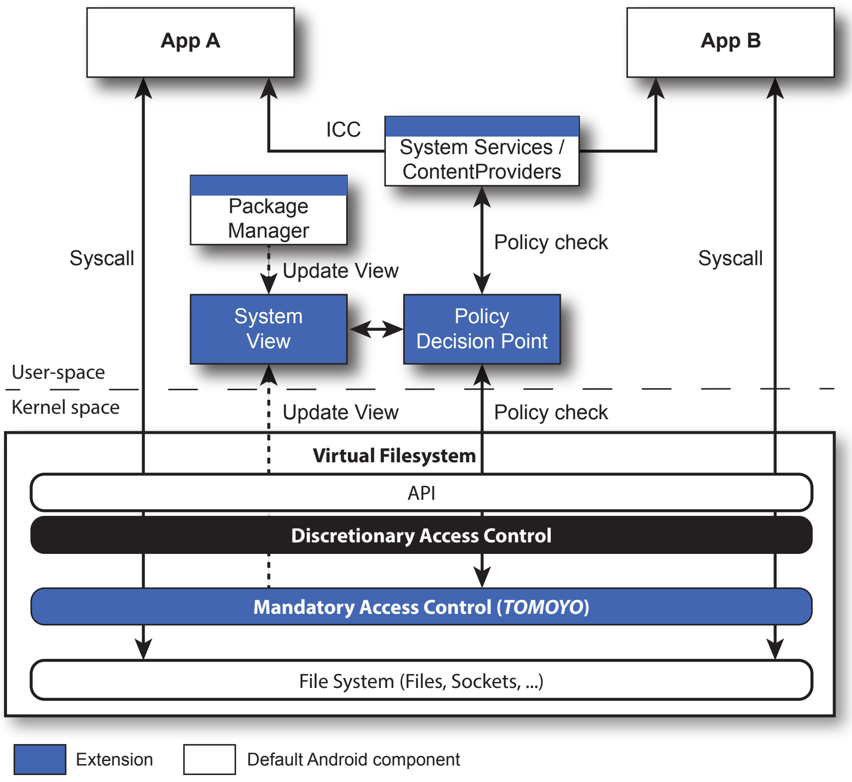

Figure 4.1: Framework architecture.

4.5.2 Architecture Components

Our security framework architecture is depicted in Figure 4.1. In the following, we explain in more technical depth the components that form our security framework and how they interact with each other. We structure our explanation in accordance with the Android software stack and first explain our middleware extensions (Section 4.5.2.1) before then explaining our kernel security extensions (Section 4.5.2.2). Lastly, we briefly explain our graph-based system view and policies (Section 4.5.2.3).

We implemented our security framework based on the Android v2.2.1 code base.

4.5.2.1 ICC Reference Monitoring

Direct ICC. We extended the Android permission check API of the *ActivityManagerService*, which controls whether an application is allowed to access the components of another application or of a system *Service/ContentProvider*, with an additional authorization hook for our framework. In case the default Android permission check would allow the inter-component communication to proceed, our new authorization hook queries the *Policy Decision Point* for a policy-based access control decision. This decision by the *Policy Decision Point* overrules the default permission check and causes the authorization hook to deny any policy violating ICC from being established. If the checked ICC is allowed by the policy, it will proceed, and the *Policy Decision Point*

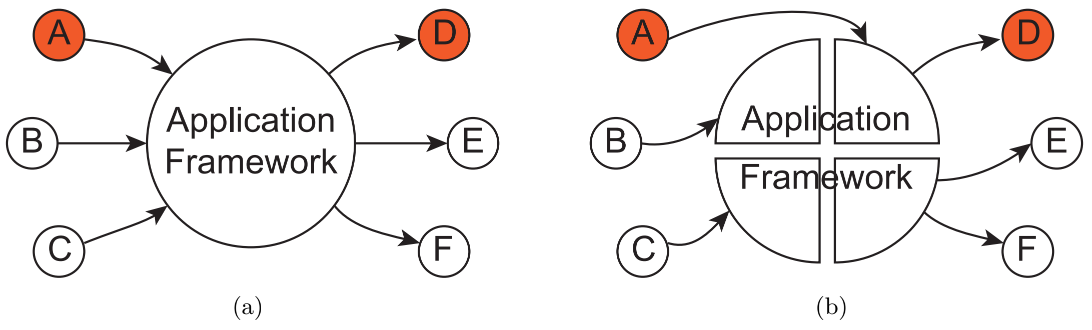

Figure 4.2: Introducing authorization hooks per system component: (a) Enforcing policies purely based on UID causes transitive closure between apps through the application framework; (b) Enforcing policies specifically for each system *ContentProvider*/*Services* of application framework avoids transitive closure. Shaded nodes represent colluding applications.

feeds this decision back to the *System View* to inform it about a new communication link in the system-wide view of established inter-application communication channels. For the policy decision, the authorization hook provides the *Policy Decision Point* with the UIDs of the caller and callee application of the to be established ICC; the *Policy Decision Point* can retrieve further information about those UIDs (e.g., their permission sets) from the system's package management.

In addition to the authorization hook in the permission check, we address direct ICC via *Broadcast Intents* by hooking the broadcast subsystem of the *ActivityManagerService*. We re-use the approach presented in our TrustDroid work (Section 3.5.1.2 on page 24) and check each sender-receiver pair. We skip sending the *Broadcast Intent* to *Receivers* that would form a policy violation in case they would receive the *Broadcast*.

Indirect ICC. A particular problem that had to be solved when monitoring inter-component communication based only on extending the permission check API are indirect information flows via the Android system *Services* and *ContentProviders* of the application framework. A majority of those components are executed under shared UIDs. Thus, using the UID from the permission check for a policy decision does not allow the *Policy Decision Point* to distinguish between the different system components. Moreover, since they implement the application API, they are commonly contacted by any installed application. As a consequence, when the *Policy Decision Point* tries to connect two different interactions of distinct apps with the system components in order to detect collusion attacks, it easily results in a transitive closure between all installed apps (see Figure 4.2(a)).

To solve this issue, we extended the calls to the permission check API in those system components, such that they provide an additional unique identifier to the *Policy Decision Point*, which the *Policy Decision Point* uses to identify the system component from which a permission check originates. Thus, the *Policy Decision Point* can now distinguish through which system component which flows can be connected and hence avoids the risk of transitive closure among all apps (see Figure 4.2(b)).

Moreover, we followed the approach for fine-grained filtering of data in *ContentProviders* that we introduced in TrustDroid (see Section 3.5.1.2 on page 24) and enforce access control on indirect flows via *ContentProviders* at the granularity of database entries.

Intent Tagging. An important feature for fine-grained analysis of communication links that can lead to confused deputy attacks is to establish relations between different ICC calls and/or Intents. Inspired by the solution of QUIRE [36], our framework builds a call-chain of the UIDs in related ICCs. In contrast to QUIRE, we opted for a system-centric call-chain by automatically tagging newly created *Intents* with the UID of the sending application. This information is used by the *Policy Decision Point* to re-create the path in the graph (which is part of the *System View*) that lead to the current ICC. A more fine-grained approach to establish call-chains based on extensions to the Binder IPC kernel module is presented in our Scippa solution (see Chapter 7).

4.5.2.2 File System Reference Monitoring

To enable mandatory access control at the kernel level, our implementation employs *TOMOYO Linux*[1] v1.8, a path-based MAC implementation available as a kernel patch. Whenever an application process creates or accesses a file or socket, TOMOYO intercepts this operation and checks with its policy whether the operation is permitted. However, since in our security framework the policy decision is dependent on the access control decision history of applications and as such is highly dynamic, TOMOYO's static policies are inadequate. To enable such highly dynamic policies on TOMOYO, we leverage the readily available user-space interface of TOMOYO and extend it in order to provide a feedback channel between the kernel and the *Policy Decision Point* and *System View* at middleware. To enable communication between the two layers, we wrote a native library with access to TOMOYO's interfaces and compiled it against the Java Native Interface (JNI). In order to enable runtime policy updates, we extended an interface of TOMOYO that allows it to seek a decision from a *supervisor* (e.g., our *Policy Decision Point*). To enable the middleware to make a decision at runtime, TOMOYO passes on the UID of the process making a system call along with details pertaining to the call itself, e.g., path and owner UID of the file to be read. The middleware's *Policy Decision Point* processes the request and conveys its decision to TOMOYO. In case TOMOYO reports that a file/socket is being created/removed, the *System View* is updated accordingly.

It is important to note that the TOMOYO kernel boots with a carefully written security policy catered for Android. This policy file is TOMOYO specific and is loaded into the kernel memory during device boot. On intercepting a system call post device boot, TOMOYO inspects its internal policy file to see if there is a policy rule that allows the system call to proceed normally. If yes, the request is granted *without* querying the *Policy Decision Point* (in the middleware). If no, TOMOYO queries the *Policy Decision Point* for a decision on the request made. If the *Policy Decision Point* deems it safe to grant the request, it conveys the same to TOMOYO and also updates the *System View* with the new communication link. There are two ways in which the *Policy*

[1] `http://tomoyo.sourceforge.jp/`

Decision Point could relay a *grant* decision: (1) It could simply request TOMOYO to allow the specific system call to proceed normally, or (2) it could in addition to (1) request TOMOYO to add the decision to TOMOYO's policy file. Such a flexibility allows our framework to reduce the number of context switches between the middleware and the kernel, which in turn reduces the performance overhead that could stem from frequent context switches.

An alternative to TOMOYO could be SELinux [114], a type-based MAC implementation available as a Linux Security Module (LSM). However, SELinux enforces MAC by means of type enforcement, which typically requires extended file attributes to be enabled in the filesystem. Since Android v2.2.1's flash file system (YAFFS2) does not support extended file attributes by default, using SELinux requires prior file system modifications. On the other hand, TOMOYO, being a path-based MAC implementation, does not require any file system modifications prior to usage. Furthermore, SELinux lacks an interface for efficiently updating the policies, thereby precluding communication between the kernel and the middleware as we require them in our framework.

4.5.2.3 Graph-Based System View and Policies

The *System View* component of our framework stores the history of all granted communication channels and thus provides a system-wide view of the communication links between applications. We opted for storing this history in form of a graph-based representation, where nodes represent applications in the system and edges represent the different kinds of monitored communication links (i.e., ICC, file read/write, socket read/write). Inherently, this representation can only be as accurate as the reference monitors that provide feedback on the communication channels. For our framework, this means that the graph-based system view operates at the granularity of UIDs, meaning at application sandbox level. Thus, apps with shared UID are represented by the node the system view graph. The exception from this limitation are the system components, which can be represented at a better granularity per component due to the unique identifier that permission checks within those system components provide to the *Policy Decision Point* and *System View*. For implementation of our graph, we chose the *JGraphT*[2] library in version 0.7.3.

The system view graph is initiated during first boot of the system and nodes for all existing apps are added to the graph including meta-information such as their UID, permissions (in case of apps), or path and owner UID (in case of files/sockets). For shared UIDs, the information of all apps under the shared UID are merged into their associated node, e.g., the union set of their permissions. To add newly installed apps to the system graph or remove uninstalled apps from it, we instrumented the *Package Manager* to provide feedback about those events to the *System View*. Nodes that represent files or sockets are added/removed from the graph when the kernel MAC (TOMOYO) informs the *System View* about those operations via the feedback channel between our middleware extensions and TOMOYO.

Figure 4.3 provides a simple example for our system graph with exemplary communication links. Besides a number of application sandboxes, it also illustrates distinct files

[2]`http://www.jgrapht.org/`

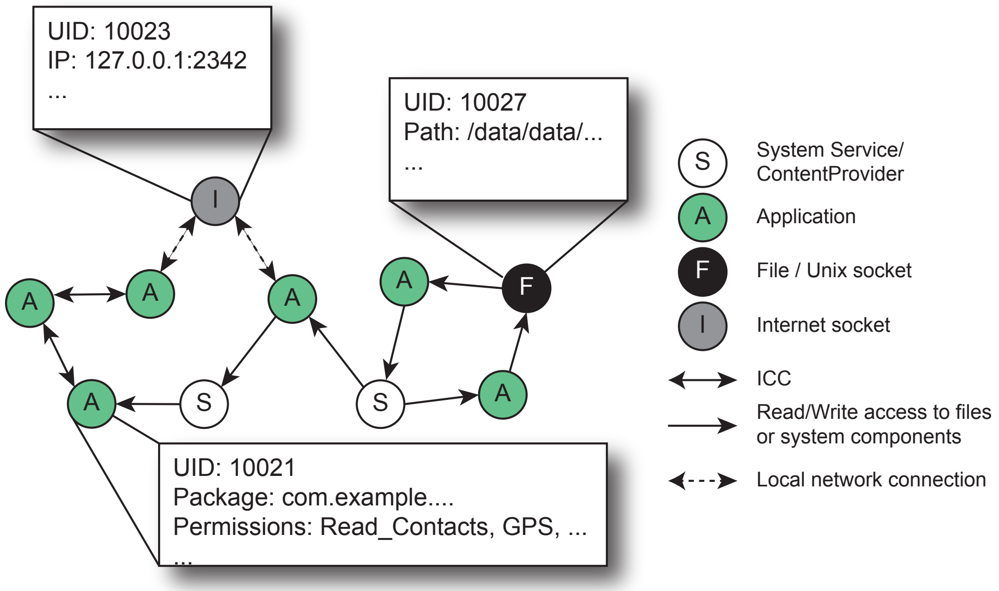

Figure 4.3: System graph representation with example communication channels.

and sockets as well as single system *Service* and *ContentProvider* components. Moreover, TOMOYO provides exact information on the data flow direction (i.e., read or write) and hence edges to/from file and Unix socket nodes can be represented unidirectional. Similarly, the authorization hooks in the system components can clearly distinguish between read and write operations, thus allowing edges to/from system components nodes to be represented unidirectional. In contrast, ICC between applications and to/from Internet sockets require a bidirectional representation for different conceptual and technical reasons: First, both ICC and Internet connections establish very often inherently a bidirectional communication flow and an exact differentiation of the current case is not possible at the level of abstraction at which our framework operates. Second, in the specific case of ICC, mechanisms, such as *Pending Intents* obfuscate the exact data flow, thus, we conservatively assume bidirectional data flows.

Operating with a graph-based representation allows to more efficiently trace (potential) communication links between applications and, thus, define our security policies using graph-based policy languages. For our implementation, we adopted the VALID [15] language for virtualized environments to our Android-specific use-case and extended it with new statements that allow us to define attack states as paths in the system graph based on the nodes' and edges' properties. For more details on our policy language and how it is used, we refer to our publication [P5, T4]

4.6 Evaluation

We begin this section with a heuristic study of communication patterns between 3^{rd} party applications that, in part, motivated our design decisions. Subsequently, we provide test results on effectiveness and performance of our framework and discuss challenges and problems therein.

4.6.1 Test methodology

As methodology for our evaluation, we opted for manual testing with a group of 25 test users, as automated testing of mobile phone applications has been shown to exhibit a very low execution path coverage (approximately 40% in average and only 1% in worst case [65]). In light of this limitation, we argue that 50 selected applications from different market categories (e.g., games or social tools) form a representative testing set. The test users' task was to install and thoroughly use the provided apps, to trigger as much as possible of the apps' features and with interleaving installation, uninstallation, and usage.

4.6.2 Study of 3^{rd} Party Application Communication

We performed a heuristic analysis of the communication patterns between third party applications from the Android Market. In the following, we present our main observations and thereby motivate our design decisions.

File system and socket based communications. Figure 4.4 depicts the file system and socket based communication between the test apps. We can observe that the applications we tested neither share their data with other applications at the file system level, nor communicate with each other via Unix domain sockets. Consequently, an attack vector that uses files or Unix sockets as communication medium could be easily identified, making it easier to prevent such an attack. To effectively prevent this attack vector, a kernel-level MAC is required, since applications can make use of native code that circumvents any security mechanisms in Android's middleware or app runtime.

Moreover, since legitimate applications are far less likely to communicate this way, the rate of falsely denied communication links is expected to be low. To illustrate this point, we developed sample apps that use a Unix socket or a file for communication and this pattern is clearly distinguishable from other applications in Figure 4.4 (bottom left corner).

ICC based communication. Our observation regarding ICC based communication shows that applications usually operate autonomously and do not have functional interdependencies with other applications. Exceptions are custom launcher applications, which start the *Activities* of other apps, and apps with a *"share with"* functionality that receive data from other apps via Intent for processing (e.g., Facebook, Twitter etc.). The usual way for apps to share data are (system) *ContentProviders*.

Our design accurately addresses this communication pattern, since (1) it implements a very fine-grained policy enforcement in the system components, and (2) direct communication between apps, which is the main target of generic system policy rules, occurs seldom and if so, with a very distinct pattern (*start Activity* or *share with*).

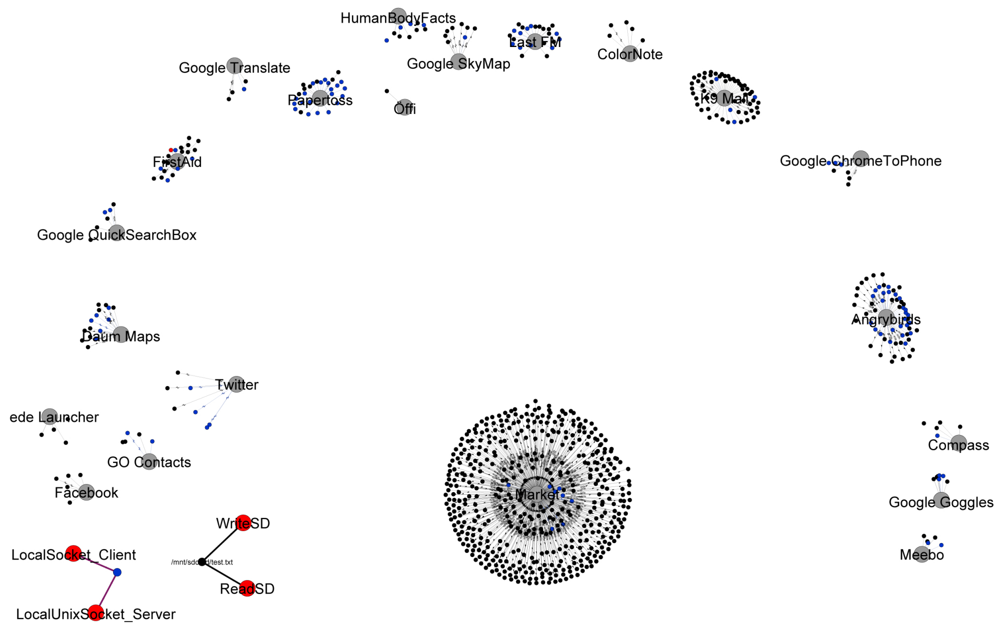

Figure 4.4: Visualization of file system and socket access by third party apps. Grey nodes represent benign applications, red nodes represent colluding applications, black nodes represent files and blue nodes represent sockets. On the bottom left are two examples of colluding applications: one pair of apps (WriteSD and ReadSD) colludes by means of a shared file on the SDcard, while another pair (LocalSocket_Client and LocalUnixSocket_Server) colludes using a socket in a client-server model.

4.6.3 Effectiveness

We evaluated the effectiveness of our solution based on the detection rate of attacks specified in an example security policy (i.e., the *false negative* rate) and the rate of falsely denied communications between applications (i.e., *false positive* rate). An ideal solution would provide both a zero false negative and zero false positive rate. Our current instantiation applies over-approximation of communication links, except for Intents (cf. Section 4.5.2.1). This means that it assumes a relation between communication channels where none might exist, e.g., it has no false negatives but tends to cause false positives.

Attack detection rate. To evaluate the detection rate of privilege escalation attacks, we developed sample applications that implement the attacks described in [161, 148, 44, 30] and deployed a system policy that contained rules targeting these attacks (we refer to our paper [P5] for details on the rules). Our framework successfully detected all of the above mentioned attacks at ICC and file system level, including all the attack scenarios of Soundcomber [161] launched via covert channels in Android system components and a file based covert channel. However, it must be noted, that the prevention of illegal channels depends on the deployed policies and other channels than the ones tested (e.g., using file sizes as covert channel), would require definition of corresponding policies.

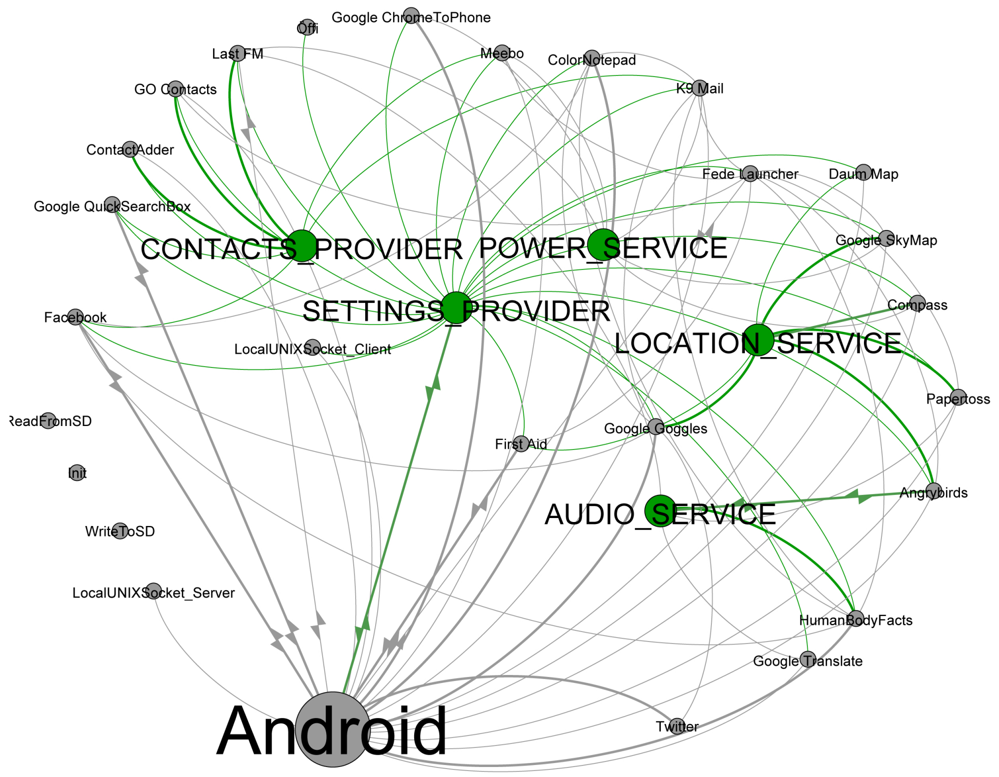

Figure 4.5: Visualization of the ICC based communication during user tests. Selected System Services and Content Providers are illustrated as separate nodes.

Falsely denied communication rate. We evaluated with our user test the impact of our security framework on the usability of third party applications. During this test, in addition to the policy rules from the attack detection test, we deployed generic rules to prevent the leakage of sensitive information like the device location, contacts, or SMS via the Internet. Moreover, we deployed a singular rule that allows applications to launch other applications with an Intent if and only if the Intent does not contain any additional data/information.[3]

To our surprise, the results of our tests showed no false positives. This is on the one hand counter-intuitive, since our policy rules also included rather generic rules and are indicative of a higher rate of false positives. On the other hand, this result confirms our observations and conclusions on the communication patterns between third party applications on Android.

4.6.4 Performance

Our solution imposes only a negligible runtime overhead, not perceivable by the user. Tables 4.1 and 4.2 present our benchmark results. In particular, as Table 4.1 shows,

[3]Note, data-less Intents can be used by the adversaries to establish covert communication. Thus, the rules should include less generic exceptional rules which, e.g., additionally specify application names.

Type	Calls	Average (ms)	Std. dev. (ms)
Original Reference Monitor runtime for ICC			
system	11,003	0.184	2.490
***Policy Decision Point* overhead for ICC**			
uncached	312	6.182	9.703
cached	10,691	0.367	1.930
Intents	1,821	8.621	29.011
***Policy Decision Point* overhead for file read**			
file read	389	3.320	4.088

Table 4.1: ICC timing results.

Type	Average (ms)	Std. dev. (ms)
Read access to System Content Providers		
total number of accesses: 591		
read	10.317	41.224
overhead	4.983	36.441
Read access to System Services		
total number of accesses: 87		
read	8.578	20.241
overhead	0.307	0.4318

Table 4.2: Timing results for system components.

our framework performs quite steadily in terms of a very low standard deviation; the runtime system call latency is low considering the ratio of cached to uncached decisions. Only the overhead on *Intent* messages cannot be optimized through caching. However, in follow-up work [P1] we implemented a higher efficient and faster system-centric ICC call-chaining based on modifications to the Binder mechanism instead of only Intents.

On read access to system *ContentProviders* (see Table 4.2), the filtering of values conflicting with the system policy imposes an overhead of approximately 48%. On read access to system *Services*, the overhead is merely about 2.4% on average. This discrepancy stems from the fact, that on read access to *ContentProviders* usually multiple reader-writer pairs have to be checked, while access to system *Services* involves only one reader-writer pair.

4.6.5 Impact on 3rd party applications' usability

Although we did not observe any false positives in our tests, any falsely denied communications must be avoided, because they can have a severe impact on the usability of the smartphone. Denying applications ICC that was expected to be successful most likely renders these application dysfunctional. Moreover, application developers do not anticipate this situation, since installed applications have been granted all the requested permissions, and thus often omit exception handling code, causing applications to crash in case their ICC call is denied.

This problem applies to all approaches based on revoking permissions at post-install time or on denying ICC at runtime, as we will explain in Section 4.7 on related work. It is particularly hard to solve for situations where one cannot clearly distinguish between a confused deputy attack and a legitimate user action, e.g., when an app that provides a “share with” functionality receives an *Intent* to share data or when the browser app is called to open a particular URL.

In our solution, we strive for minimizing the number of false positives by (1) decomposing the rather monolithically structured application framework sandboxes into distinct *Services* and *ContentProviders* nodes in our system graph; and (2) by performing a heuristic analysis of the communication patterns of applications. The former one facilitates a more fine-grained policy check on access to data shared via system components, while the latter is used to refine the policy design.

4.7 Related Work

In this section we provide an overview of closest related work on extending Android’s security framework. We first discuss in Section 4.7.1 the related work at the time this work was conducted in order to underline the novelty of our results. We then discuss in Section 4.7.2 further related work that has been published after our work had been presented.

4.7.1 Status Quo at Time of Publication

Android Security Extensions. Several security extensions to Android’s security framework have been proposed prior to our work [44, 140, 166, 133, 46, 24, 139, 149]. In the following paragraphs, we discuss the strengths and shortcomings of security extensions that are closest to ours.

The goal of *Kirin* [44], one of the first security extensions for Android, is to mitigate malware contained within a single application. It extends the Android application installer to check at install-time if an application requests an undesired combination of permissions and denies installation if so. Additionally, Kirin is capable of determining which interfaces of other applications an application can potentially access. However, due to its static nature, Kirin has to conservatively consider all potential communication links as possible, in particular all unprotected interfaces must be assumed as being accessed. This over-approximation will eventually stop any application from being installed, since applications can potentially establish arbitrary communication links to other applications. In contrast to Kirin, our framework (1) focuses on runtime monitoring of actual communication links and establishing connections between them; and (2) decides in real-time if a new link is allowed to be established or would violate the security policy.

Saint [140] allows developers to attach policies to their apps that govern how other apps can interact with their apps. The policies are enforced by different authorization hooks in crucial system services such as *ActivityManagerService*. In order to prevent confused deputy attacks with Saint, app developers have to define appropriate security policies for each of their app’s interfaces, i.e., they have to specify which permissions/-configuration/signature a calling app is required to have in order to access an interface. However, since application developers have to define these policies themselves, they might fail to consider all security threats. Finally, Saint does not address malicious developers, who will not deploy Saint policies for the obvious reason that they might want to mount a collusion attack. By contrast, our framework deploys a system-centric solution which is also applied to malicious colluding applications, and enforces its policies at the file system and network layer as well.

Quire [36] provides a lightweight provenance system to prevent confused deputy attacks via Binder IPC. It tracks and records the IPC call chain across multiple and allows receivers of IPC to inspect the call chain in order to evaluate their trust in the originator of the IPC call. However, similar to Saint, Quire is application-centric. It requires applications to explicitly forward the IPC call chain to subsequent apps in the communication chain. Thus, Quire cannot prevent colluding applications, because they may drop the IPC call chain and hide their partner app.

IPC Inspection [148], like Quire, also tackles confused deputy attacks. It poly-instantiates applications and reduces the permission set of an application instance that receives a message from a less privileged app, so that the receiving instance cannot be used for escalating the sender's privileges. This can be considered as a relaxed instantiation of the Biba [13] and LOMAC [57] integrity models by lowering the integrity level (i.e., permission set) of the app that receives the message to the integrity level of the sending app. In contrast to our framework, IPC Inspection does not require a policy framework, and hence, can prevent unknown attacks without the deployment of appropriate policies. However, IPC Inspection does not provide a solution against maliciously colluding applications. Although the receiver instance's permissions are reduced to the sender's permissions, the individual application instances at the receiver's side still reside in one sandbox and are not properly isolated from each other. Thus, a malicious application can simply let their individual instances collaborate within its sandbox and thus circumvent the permission reduction. An important question that the authors of IPC inspection left open is how permissions that are controlled by the underlying Linux kernel (e.g., Internet or bluetooth) are effectively revoked at runtime. Lastly, reducing the receiver's permissions to the ones of the sender contradicts Android's privilege separation model: on Android, applications *should* refrain from asking for too many permissions and instead delegate certain tasks to sufficiently privileged applications.

TaintDroid [46] and *AppFence* [93] use dynamic taint tracking of data from security and privacy sensitive sources to detect [46]/prevent [93] unauthorized leakage of sensitive data via selected sinks (such as Internet sockets). Both are able to detect data leakage attacks potentially initiated through a application-level privilege escalation attack. However, TaintDroid mainly addresses data flows, whereas privilege escalation attacks also involve control flows. TaintDroid's authors mention that tracking the control flow with their system will likely result in much higher performance penalties. AppFence additionally provides access control to system *ContentProviders* and *Services*, where it introduces *data shadowing* to tackle the problem of crashing applications when permissions are dynamically revoked (e.g., by returning fake or blank data). Nevertheless, like TaintDroid, AppFence provides no means to detect privilege escalation attacks beyond data leakage attacks.

SELinux on Android [166] presented the first implementation of SELinux on Android and argues for the benefits SELinux can provide in securing Android-based devices. Although SELinux later became an integral part of Android's security architecture, SELinux is not a suitable solution for our framework. As we argued in Section 4.5, SELinux lacks a coordination mechanism between the Linux kernel and the Android middleware. However, such a coordination mechanism is crucial in bridging the semantic gap between those two layers in order to adequately address collusion and confused deputy attacks, which leverage communication channels at both middleware and kernel level. We addressed this challenge with our TOMOYO-based architecture.

Our XManDroid work was preceded by our TrustDroid work [P4] (see previous Chapter 3). While TrustDroid and XManDroid share a common design pattern (e.g., enforcement at both kernel and middleware level), TrustDroid is designed for domain isolation based on a static (and hardcoded) policy. For XManDroid we required a more dynamic security enforcement and hence also a more flexible interaction between kernel and middleware level (see also requirements analysis in Section 4.4 on page 43). Thus, while TrustDroid formed a foundation of this work, it required far-reaching extensions.

Non-Android Related Work. The XManDroid framework presented in this work applies results from former research on operating system security. First, our work relates to *stack inspection* (e.g., [191]), a security mechanism that enforces access control decisions by inspecting the runtime call stack of a request, by building call-chains across applications (similar to Quire [36], however, in a system-centric approach).

Moreover, our framework is related to the *Chinese Wall* security model [18], where the history of granted access control requests of a subject determine the result of all possible current and future access control decisions for this subject. The overall goal of this model is to prevent information flow between subjects sharing the same conflict of interest class. Similarly, our framework makes access control decisions based on which components or files/socket an application sandbox has accessed in the past.

Furthermore, XManDroid's filtering of ICC that implements known confused deputy attacks could also be abstractly considered as a centralized instantiation of the *Clark-Wilson integrity model* [22], which requires processes of higher integrity level to accept input from lower-integrity processes only via filtering interfaces. The integrity level of an application depends in this case directly on the permission set that application holds.

4.7.2 Related Work Post-Publication

Since publication of our XManDroid solution, a large body of related work has been established. In the following, we survey selected closest related work.

Various works have identified new forms of confused deputy attacks [215, 125, 52]: First, is was discovered, that application developers tend to leave their component interfaces unprotected, allowing an unprivileged app to connect to those components and *leak their content* or *pollute their content* [215]. For instance, it allowed an attacker to extract information such as user's contacts, instant messenger chat logs, or login credentials. Although XManDroid's original architecture could not detect such illegal access to unprotected application components,[4] our heuristics could be extended to cover those cases: Accessing a component of another application is a quite distinctive behavior from the usual inter-application data sharing via *Intents* ("share with") and hence might allow XManDroid to be adapted to this new attack scenario. Second, independent work [125] discovered that some Android systems that had been customized by vendors, contained a highly-privileged *Service* component confused deputy that effectively offered a root shell to any application (via *Intents*). Given an appropriate policy, XManDroid could protect this vulnerable system *Service*. Lastly, the Android *Clipboard* service was found to be exploitable as confused deputy [52]. Since the *Clipboard* is unprotected, any application can monitor and retrieve the clipboard content. When the user now copies sensitive data between apps, e.g., a password from a password manager app to the browser, an attacker app can retrieve this information as well. The *Clipboard* service operates semantically similar to *ContentProviders* and hence XManDroid's per-data enforcement could be easily extended to the *Clipboard* service.

Very recently, the need for precise inter-application ICC classification for preventing collusion attacks has been emphasized [43]. The authors of that work have re-evaluated the policies we used in our work for preventing collusion and confused deputy attacks (see [P5] for details) and concluded that in today's Android application model the false positive rate of our enforcement would in fact be noticeably higher. As an improvement of our work, the authors propose an approach based on in-depth static flow analysis (similar to *Epicc* [137]) to detect cross-app data flows.

Since our framework design extends the default Android permission check API, any policy denial for ICC will result in a denied permission check (with exception of our per-data access control in the system *Services* and *ContentProviders*). Besides AppFence, a more recent work [103] has quantified the negative effects such denials have on 3^{rd} party applications. This motivates the need for a more graceful enforcement of denied ICC in a future version of our framework.

Further, the attacker model for Android has been updated since this work was presented. It was discovered that in many cases not the app developer but instead the provider of external libraries is using dubious privacy practices—foremost providers of advertisement libraries [47, 84]. However, Android's security model does not make a distinction between the external lib as security principal and its host app as security principal. Thus, our design is currently limited to enforcing policies on the application sandbox as a whole and not for the security principals that actually form the application code. Approaches that retrofit Android's application model accordingly, and which hence would greatly improve the effectiveness of our XManDroid, are *AdSplit* [170] and *AdDroid* [144]. Another promising approach is *Compac* [196], which allows the system to enforce per app-component permissions.

Similarly to TrustDroid, *MOSES* [156] provides domain isolation between different domains such as work or private at the different layers of Android's software stack. In contrast to TrustDroid, it includes the TaintDroid [46] framework to track information flows and prevent cross-domain data leakage. However, like TrustDroid, MOSES lacks the flexibility for policy enforcement that we need in XManDroid.

[4]Remember, that is was based on extending the permission check API of the application framework and that unprotected interfaces do not trigger a check.

Aquifer [131] dynamically constructs the user interface workflow and enforces export restrictions on data (such as office documents, voice or written notes, and images) that are passed along the workflow across the involved applications in order to prevent accidental information disclosure. As such, Aquifer could limit the impact of collusion attacks, when colluding apps are involved in a user interface workflow (e.g., data intermediaries for *"share with"* functionality).

Lastly, the XManDroid scenario has been implemented as a security module on top of our Android Security Framework [P2] (see Chapter 6).

4.8 Conclusion

In this work, we addressed the problem of confused deputy and collusion attacks on Android. We propose the design and implementation of a practical security framework for Android that monitors at runtime the application communication channels in Android's middleware and in the underlying Linux kernel (namely, IPC, file system, Unix domain and Internet sockets). Our framework ensures that they comply with a graph-based security policy that prevents different classes of confused deputy and collusion attacks. To underline our design decisions, we analyzed typical communication patterns of Android applications by means of a heuristic study. Our design addresses our observations, as it implements a fine-grained policy enforcement in the system components, which are the primary means for applications to share data. Inspired by Quire [36], we integrate *Intent* tagging techniques into our system (but in a system-centric way) in order to increase the precision of our analysis. Moreover, a novelty of our prototype is the runtime interaction between our security extensions to the Android middleware and TOMOYO Linux at kernel-level, allowing for dynamic runtime policy mapping from the semantically rich middleware to the kernel access control. Our evaluation results show that our framework is efficient and effective. It can prevent recently published application-level privilege escalation attacks [44, 30, 148, 161], including sophisticated ones, such as Soundcomber [161], which uses covert channels in the Android system components.

5 FlaskDroid

Flexible and Fine-Grained Mandatory Access Control on Android for Diverse Security and Privacy Policies

5.1 Motivation

Mobile devices such as smartphones and tablets have become very convenient companions in our daily lives and, not surprisingly, also appealing to be used for working purposes. On the down side, the increased complexity of these devices as well as the increasing amount of sensitive information (private or corporate) stored and processed on them, from user's location data to credentials for online banking and enterprise VPN, raise many security and privacy concerns. Today the most popular and widespread smartphone operating system is Google's Android.

Android's vulnerabilities. Android has been shown to be vulnerable to a number of different attacks such as malicious apps and libraries that misuse their privileges [216, 146, 84] or even utilize root-exploits [217, 146] to extract security and privacy sensitive information; taking advantage of unprotected interfaces [21, 19, 211, 111] and files [177]; confused deputy attacks [30]; and collusion attacks [161, 117].

Solutions. On the other hand, Android's open-source nature has made it very appealing to academic and industrial security research. Various extensions to Android's access control framework have been proposed to address particular problem sets such as protection of the users' privacy [46, 93, 24, 210, 7, 101]; application centric security such as *Saint* enabling developers to protect their application interfaces [140]; establishing isolated domains (usage of the phone in private and corporate context) [P4]; mitigation of collusion attacks [P5], and extending Android's Linux kernel with Mandatory Access Control [132].

Observations. Analyzing the large body of literature on Android security and privacy one can make the following observations:

The first observation is that almost all proposals for security extensions to Android constitute mandatory access control (MAC) mechanisms that are tailored to the specific semantics of the addressed problem, for instance, establishing a fine-grained access control to user's private data or protecting the platform integrity. Moreover, these solutions fall short with regards to an important aspect, namely, that protection mechanisms operate only at a specific system abstraction layer, i.e., either at the middleware (and/or application) layer, or at the kernel-layer. Thus, they omit the peculiarity of the Android OS design that each of its two software layers (middleware and kernel) is important within its respective semantics for the desired overall security and privacy. Only few solutions consider both layers [P5, P4], but they support only a very static policy and lack the required flexibility to instantiate different security and privacy models.

The second observation concerns the distinguishing characteristic of application development for mobile platforms such as Android: The underlying operating systems provide app developers with clearly defined programming interfaces (APIs) to system resources and functionality—from network access over personal data like SMS/contacts to the onboard sensors. This clear API-oriented system design and convergence of functionality into designated service providers [212, 126] is well-suited for realizing a security architecture that enables fine-grained access control to the resources exposed by the API. As such, mobile systems in general and Android in particular provide better opportunities to more efficiently establish a higher security standard than possible on current commodity PC platforms [105, 212].

5.2 Challenges and Our Goal

Based on the observations mentioned above, we aim to address the following challenges in this work: (1) Can we design a generic and practical mandatory access control architecture for Android-based mobile devices, that operates on both kernel and middleware layer, and is flexible enough to instantiate various security and privacy protecting models just by configuring security policies? More concretely, we want to create a generic security architecture which supports the instantiation of already existing proposals such as *Saint* [140] or privacy-enhanced system components [218], or even new use-cases such as a *phone booth (or kiosk) mode.* (2) To what extent would the API-oriented design of Android allow us to minimize the complexity of the desired policy? Note that policy complexity is an often criticized drawback of generic MAC solutions like SELinux [114] on desktop systems [212].

5.3 Contributions

In this work, we presented the design and implementation of a security architecture for the Android OS, called FlaskDroid, that addresses the above mentioned challenges. Our design is inspired by the concepts of the *Flask* architecture [179]: a modular design that decouples policy enforcement from the security policy itself, and thus provides a generic architecture where multiple and dynamic security policies can be supported by the system. In particular, our contributions are:

System-wide security framework: We present an Android security framework that operates on both the middleware and kernel layer. It addresses many problems of the stock Android permission framework and of related solutions, which target either the middleware or the kernel level. We base our implementation on SE Android [175, 132], which has already been partially merged into the official Android source-code by Google [73, 74, 75, 76].

Security policy and type enforcement at middleware layer: We developed and integrated type enforcement at Android's middleware layer and its synchronization with the kernel enforcement at run-time. This is not trivial because the middleware layer has a completely different semantic than the kernel level. We present our policy language, which is specifically designed for the rich Android middleware semantics.

Use-cases: We show how our security framework can instantiate selected use-cases. The first one is an attack-specific related work, the well-known application centric security solution *Saint* [140]. The second one is a privacy protecting solution that uses fine-grained and user-defined access control to personal data. We also mention other useful security models that can be instantiated with FlaskDroid.

Efficiency and effectiveness: We successfully evaluate the efficiency end effectiveness of our solution by testing it against a testbed of known attacks and by deriving a basic system policy which allows for the instantiation of further use-cases.

5.4 Primer on SELinux and SE Android

5.4.1 SELinux

Security Enhanced Linux (SELinux) [114] is an instantiation of the Flask security architecture [179] and implements a policy-driven mandatory access control (MAC) framework for the Linux kernel. In SELinux, policy decision making is decoupled from the policy enforcement logic. SELinux uses the Linux Security Module (LSM) [206] architecture, which provides various access control enforcement points for low-level resources, such as files, IPC, or memory protection. When an LSM hook is triggered (e.g., a file is opened), the SELinux LSM enforces policy decisions requested from a *security server* in the kernel. This security server manages the policy rules and contains the access decision logic. Depending on the security server's decision, the SELinux security module denies or allows the operation to proceed. To maintain the security server (e.g., update the policy), SELinux provides a number of user space tools.

Access Control Model. SELinux supports different access control models such as *Role-Based Access Control* (RBAC) and *Multilevel Security* (MLS). However, *Type Enforcement* (TE) is the primary mechanism: each object (e.g., files, IPC) and subject (i.e., processes) is labeled with a *security context* containing a *type* attribute that determines the access rights of the object/subject. By default, all access is denied and must be explicitly granted through policy rules—*allow rules* in SELinux terminology. Using the notation introduced in [85], each rule is of the form

$$allow\ T_{Sub}\ T_{Obj}\ :\ C_{Obj}\ O_C$$

where T_{Sub} is a set of subject types, T_{Obj} is a set of object types, C_{Obj} is a set of object classes, and O_C is a set of operations. The object classes determine which kind of objects this rule relates to and the operations contain specific functions supported by the object classes. If a subject whose type is in T_{Sub} wants to perform an operation that is in O_C on an object whose class is in C_{Obj} and whose type is in T_{Obj}, this action is allowed. Otherwise, if no such rule exists, access is denied. For instance, the rule

$$allow\ useradd\_t\ passwd\_t\ :\ file\ write$$

defines that a process (subject) with type *useradd_t* is allowed to *write* an object with class *file* and type *passwd_t*. This rule is important on multi-user desktop systems, where the `/etc/passwd` file contains essential user information and thus should be protected. The `useradd` tool adds a new user to the system by adding the new user's information to the `passwd` file and thus requires write access. A typical SELinux policy on Fedora Linux 17 currently defines more than 600 types, almost 100 classes, and more than 100,000 allow rules. We evaluate policy complexities of different SELinux versions and of FlaskDroid in more detail in our Evaluation Section 5.8.1.

The *user* and *role* attributes of the security context form the basis for SELinux *Role-Bases Access Control*, which builds upon type enforcement by defining which type and role combinations are valid for each user in the policy.

Optionally, SELinux supports *Multilevel Security*, which extends the fundamental type enforcement. When MLS is enabled, the security context is extended with *low security level* and *high security level* attributes (in the spirit of lattice based access control [12, 35]), where the low level represents the current security level of a subject/object and the high level represents the clearance level of the subject/object. Each security level is described by a *sensitivity* and a set of *categories*. Sensitivities are strictly hierarchical and reflect an ordered data sensitivity model (e.g., TopSecret, Secret, Unclassified) [118]. Categories are unordered and reflect data compartmentalization (e.g., Research, Human Resources, Contracts). With MLS enabled, access to objects is only allowed of the subject holds a high enough security clearance (sensitivity) and the correct category for the object.

Dynamic policies. SELinux supports to some extent dynamic policies based on *boolean flags* that affect *conditional policy* decisions at runtime. These booleans and conditionals have to be defined prior to policy deployment and new booleans/conditions can *not* be added after the policy has been loaded without recompiling and reloading the entire policy.

The simplest example for such dynamic policies are booleans to switch between "enforcing mode" (i.e., access denials are enforced) and "permissive mode" (i.e., access denials are not enforced, but at most logged). Other booleans can, for instance, refine the coverage of the access control, e.g., by enabling/disabling access control on certain object classes like files or sockets.

Technically, this mechanism is implemented in the form of *if* statements for *allow* rules in the policy. Thus, only when the *if* condition evaluates to *True*, the rules in the block of the *if* statement are considered during access control decisions.

User Space Object Managers. A powerful feature of SELinux is that its access control architecture can be extended to security-relevant user space daemons and services, which manage data (objects) independently from the kernel [192]. Thus, such daemons and services are referred to as *User Space Object Managers* (*USOMs*). They are responsible for assigning security contexts to the objects they manage, querying the SELinux security server for access control decisions, and enforcing these decisions. Prominent examples for such *USOMs* on Linux systems include the *X Window System server* [193] (Linux' display manager), *SE-PostgreSQL* [104] (a security-enhanced object-relational database system), *D-BUS* (a message bus system for inter-process communication), or *GConf* [20] (the GNOME settings manager).

Alternatively to querying the kernel space security server, USOMs could query a user space security server for access control decisions.[1] However, this approach is not any longer pursuit by the SELinux developers.

[1]http://oss.tresys.com/projects/policy-server

5.4.2 SE Android

The security benefits of integrating SELinux into the Android software stack have been first described by Shabtai, Fledel, and Elovici [166]. The authors outline how SELinux can mitigate or limit the effects of privilege escalation attacks against the critical, highly-privileged system services in the Android user space. Their work also presents a prototypical integration of SELinux into the Android OS version 1.6 and discussed many of the integration challenges that have been later tackled by the SE Android project [175] (e.g., integrating type transition for application processes into Zygote).

SE Android [175, 174] fully prototypes SELinux for Android's Linux kernel and demonstrates the value of SELinux in defending against various root exploits and application vulnerabilities on the Android platform. Specifically, it confines system *Services* and apps in different kernel space security domains even isolating apps from one another by the use of the Multi-Level Security (MLS) feature of SELinux. To this end, the SE Android developers started writing an Android-specific policy from scratch. In addition, SE Android provides a few key security extensions tailored for the Android OS. First, SE Android introduces new hooks for Android's Binder driver making the latter a *Kernel Space Object Manager*. This ensures that all Binder IPC is subject to SE Android policy enforcement. Second, it labels application processes with SELinux-specific security contexts that are later used in type enforcement. In contrast to traditional desktop platforms, where new application processes are spawned when executing a binary, on Android new app processes are forked from a system process, denoted *Zygote*, which is pre-initialized with all important shared libraries and thus enables fast starts of new app processes. Thus, security labeling had to be integrated into the Zygote mechanism in order to label the newly created app processes accordingly. Moreover, the SE Android developers had to start writing an SE Android specific policy from scratch. This new policy contained at the time this work was conducted more than 200 types, about 80 object classes, and roughly 1,400 allow rules, which is magnitudes smaller than previous SELinux policies. Thirdly, since (in the majority of cases) it is a priori unknown during policy writing which apps will be installed on the system later, SE Android employs a mechanism to derive the security context of applications at install-time. Based on criteria, such as the permissions the app requests or its developer signature, apps are assigned a security type. This mapping from application meta-information to security types is defined in the SE Android policy.

Middleware MAC. While the above listed security mechanisms are directly derived from SELinux and address the lower level of the Android software stack (e.g., files, sockets, and IPC), SE Android additionally provides *rudimentary* support for MAC policy enforcement at the middleware layer[2] (*MMAC*) inspired by various related work [172]. In particular, *MMAC* consists of three distinct mechanisms: (1) Install-time MAC, which, similar to Kirin [44], performs a policy-driven install-time check of new applications and denies installation when the application requests a defined combination of permissions; (2) Permission revocation, which is realized similar to

[2]See also `http://seandroid.bitbucket.org/MiddlewareMAC.html#middleware-mac`

existing implementations found in custom roms[3], commercial products[4], or related work [218, P4, P5]. This mechanism overrules the default Android permission check with a policy-based decision to allow/deny an application to leverage a granted permission; (3) Intent MAC, which protects with a white-listing enforcement the delivery of *Intents* to *Activities*, *Broadcast Receivers*, and *Services*. Similar mechanisms are employed, for instance, in [218, P4, P5]. However, in SE Android, the white-listing rules are based on the security type of the sender and receiver of the *Intent* message as well as *Intent* data such as the Action string.

5.5 Requirements Analysis for Android Security Architectures

5.5.1 Adversary Model

We consider a strong adversary with the goal to get access to sensitive data as well as to compromise system or third-party apps. Thus, we consider an adversary that is able to launch *software* attacks on different layers of the Android software stack.

5.5.1.1 Middleware Layer

Recently, different attacks operating at Android's middleware layer have been reported:

Overprivileged 3^{rd} party apps and libraries. Apps can threaten the user's privacy by adopting questionable privacy practices or even spyware-like behavior. For instance, popular apps like WhatsApp [202, 201], Path [54], or Facebook [51] have been publicly debated to overstep the necessary boundaries of their access to user's private data, e.g., by uploading the entire contacts database of the *ContactsProvider* instead of only the subset of contacts information required for correct app functionality.

Moreover, advertisement libraries, frequently included in 3^{rd} party apps on Android, have been shown to exploit the permissions of their host app to collect information about the user [47, 84], including privacy sensitive data such as contacts or location.

Malicious 3^{rd} party apps. In the recent past the number of mobile malware has steadily increased [186, 50]. The predominant observed malicious behavior consists of leveraging dangerous permissions to cause financial harm to the user (e.g., sending premium SMS) and exfiltrate user-private information [216, 146].

Confused deputies. Confused deputy attacks concern malicious apps, which leverage unprotected interfaces of benign, privileged system [44, 148] and 3^{rd} party [30, 215] apps (denoted deputies) to escalate their privileges.

[3]CyanogenMod (`http://www.cyanogenmod.com`)
[4]3LM (`http://www.3lm.com`)

Collusion attacks. Collusion attacks concern malicious applications that collude in order to merge their permission sets and gain a permission set that has not been approved by the user. A prominent example for a collusion attack is Soundcomber [161], where one application has the permission to record audio and monitor the call activity, while a second one owns the Internet permission. When both applications collude, they can capture the credit card number (spoken by the user during a call) and leak it to a remote adversary. Collusion attacks can be further subclassified according to the channel over which they communicate [P5, 117], e.g., overt channels like sockets versus covert channel like file locks or volume level.

Sensory malware. Sensory malware leverages the information from onboard sensors in order to derive user's privacy sensitive information. For instance, the accelerometer provides information about the movement of the phone, which can be used to infer the user input to the virtual keyboard, e.g., passwords [211, 19].

5.5.1.2 Root Exploits

Besides attacks at Android's middleware layer, various privilege escalation attacks on lower layers of the Android software stack have been reported [217, 146], which grant the attacker root (i.e., administrative) privileges and can be used to bypass the Android permission framework. For instance, he can bypass the *ContactsProvider* permission checks by accessing the contacts database file directly. Moreover, processes on Android executing with root privileges automatically inherit all available permissions at middleware layer.

It should be noted that attacks targeting vulnerabilities of the Linux kernel are out of scope of this work, since SE Android is a building block in our architecture (see Section 5.6) and as part of the kernel it is susceptible to kernel exploits.

5.5.2 Requirements

Based on our adversary model we now derive the necessary requirements for an efficient and flexible access control architecture for mobile devices. Essentially, these requirements are valid for various mobile operating systems. In this work we focus on the popular and open-source Android OS.

Access Control on Multiple Layers. Mandatory access control solutions at kernel level, such as SE Android [132] or TOMOYO [87], help to defend against or to constrain privilege escalation attacks on the lower-levels of the OS [175, 174]. However, kernel level MAC provides insufficient protection against security flaws in the middleware and application layers, and lacks the necessary high-level semantics to enable a fine-grained filtering at those layers [175, 172]. Access control solutions at middleware level [93, 24, 140, P4, P5] are able to address these shortcomings of kernel level MAC, but are, on the other hand, susceptible to low-level privilege attacks.

Thus, a first requirement is to provide simultaneous MAC defenses at the two layers. Ideally, these two layers can be dynamically synchronized at run-time over mutual interfaces [P5]. At least, the kernel MAC is able to preserve security *invariants*, i.e., it enforces that any access to sensitive resources/functionality is always first mediated by the middleware MAC and no (low-level) privilege escalation attack, such as a root exploit, bypasses the middleware access control.

Multiple stakeholders policies. Mobile systems involve multiple stakeholders, such as the end-user, the device manufacturer, app developers, or other 3^{rd} parties (e.g., the end-user's employer). These stakeholders also store sensitive data on the device. Related work, such as *Saint* [140], TrustDroid [P4], or *TISSA* [218], has proposed special purpose solutions to address the security requirements and specific problems of app developers, 3^{rd} parties (here companies), or the end-user, respectively. Naturally, the assets of different stakeholders are subject to different security requirements, which are not always aligned and might conflict. Thus, one objective for a generic MAC framework that requires handling policies of multiple stakeholders is to support (basic) policy reconciliation mechanisms [152, 120]. For instance, [152] discusses different strategies for reconciliation, such as *all-allow* (i.e., all stakeholder policies must allow access), *any-allow* (i.e., only one stakeholder policy must allow access), *priority* (i.e., higher ranked stakeholder policies override lower ranked ones), or *consensus* (i.e., at least one stakeholder policy allows and none denies or vice versa).

Context-awareness. The security requirements of different stakeholders may depend on the context the device is currently used in. For instance, the security policy of an enterprise might dictate that certain assets, such as apps, may only be accessed during working hours or while the device is located on enterprise premises. Thus, our architecture shall provide support for context-aware security policies.

Support for different Use-Cases. Our architecture shall serve as a basis for different security solutions applicable in a variety of use cases. For instance, by modifying the underlying policy our solution should be able to support different use cases (as shown in Section 5.7), such as the selective and fine-grained protection of app interfaces [140], multiple isolated security domains, e.g., dual-persona smartphones that isolate enterprise and private interests from each other [P4], or privacy-enhanced system *Services* and *ContentProviders*.

Advantage of mobile OSes for policy complexity. At first glance it may seem very challenging to realize a fine-grained and flexible access control at both the middleware and kernel-level. In particular, SELinux [114] and similar Mandatory Access Control solutions on desktop and server systems are notorious for their extremely complex policies (cf. Section 5.8.1) and a comparable complexity could be expected for our solution. However, the design of mobile operating system differs in one important point from the design of traditional platforms: Security and privacy critical functionality is concentrated in a small number of privileged system components, which expose these functionality through clearly defined interfaces to applications. This is a distinct advantage for reducing the policy complexity in FlaskDroid: (1) The privileged system components form a single point of access to their functionality and instrumenting them as MAC enforcement point achieves inherently a high degree of coverage; (2) The extent

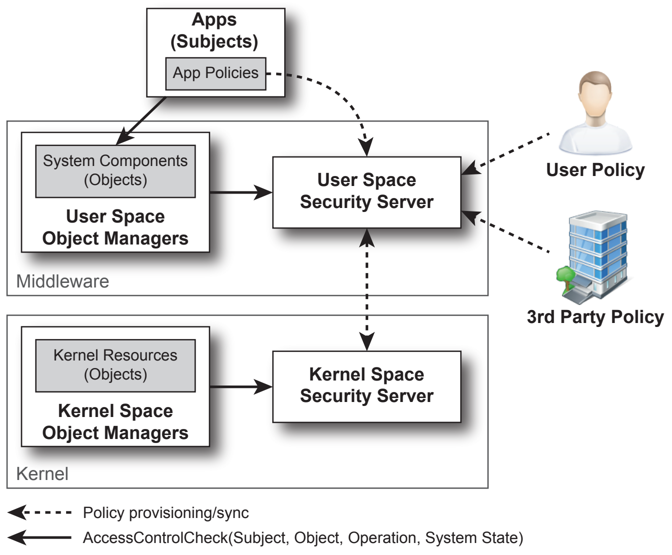
Apps (Subjects)
App Policies
System Components (Objects)
User Space Object Managers
User Space Security Server
Middleware
User Policy
3rd Party Policy
Kernel Resources (Objects)
Kernel Space Object Managers
Kernel Space Security Server
Kernel
Policy provisioning/sync
AccessControlCheck(Subject, Object, Operation, System State)

Figure 5.1: FLASKDROID concept.

of the policy primitives (e.g., object classes and operations) and rules is significantly reduced, since only a small number of system components has to be instrumented as MAC enforcement points.

5.6 FlaskDroid Architecture

In this section, we provide an overview of our FLASKDROID architecture, elaborate in more detail on particular design decisions, and present the policy language employed in our system. We implemented a prototype of FLASKDROID based on SE Android in revision 4.0.4 [132] on a Samsung Galaxy Nexus phone.

5.6.1 Overview

The high-level idea of FLASKDROID is inspired by the Flask security architecture [179], where various *Object Managers* at middleware and kernel-level are responsible for assigning their objects security contexts (see Figure 5.1). Objects can be, for instance, kernel resources such as *Files* or *IPC* and middleware resources such as *Service* interfaces, *Intents*, or *ContentProvider* data. On access to these objects by subjects (i.e., apps) to perform a particular operation, the managers enforce an access control decision that they request from a security server at their respective layer. Thus, our approach implements a *user space security server*. Access control in FLASKDROID is implemented, as in SE Android [175], as *type enforcement*. However, in contrast to SE Android we extend our policy language with new features that are tailored to the Android middleware semantics (more details follow in Section 5.6.3). Moreover, to enable more dynamic policies, the policy checks in FLASKDROID depend also on the *System State*, which determines the actual security context of the objects and subjects at runtime.

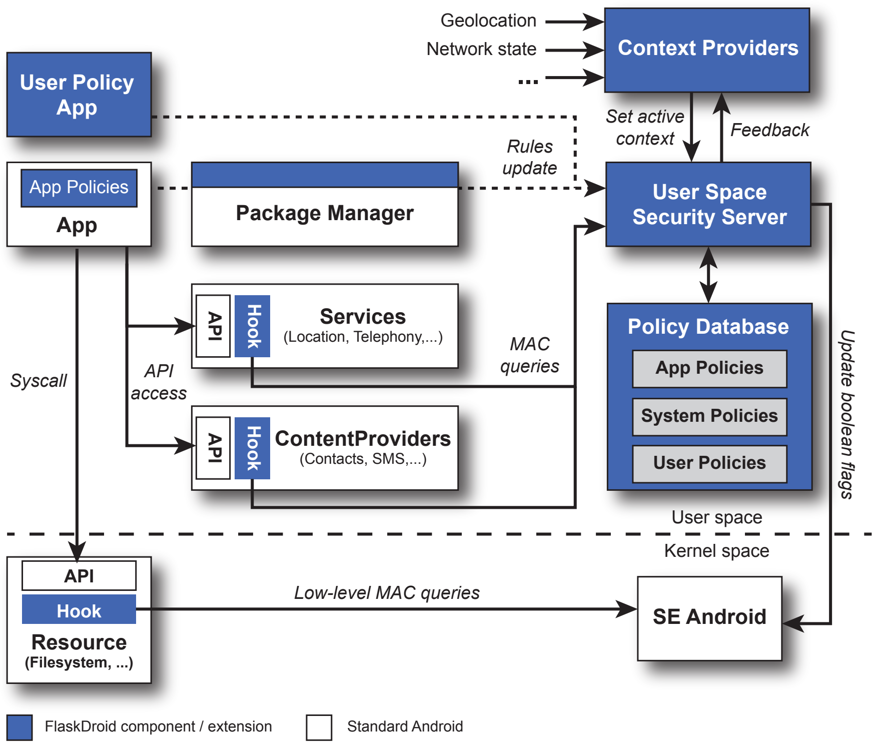

Figure 5.2: FLASKDROID architecture.

Each security server is also responsible for the policy management for multiple stakeholders such as app developers, end-user, or 3rd parties. A particular feature is that the policies on the two layers are synchronized at runtime, e.g., a change in enforcement in the middleware, must be supported/reflected at kernel-level. Thus, by decoupling the policy management and decision making from the enforcement points and consolidating the both layers, the goal of FLASKDROID's design is to provide fine-grained and highly flexible access control over operations at both middleware and kernel-level.

5.6.2 Architecture Components

Figure 5.2 provides an overview of our architecture. In the following, we will explain the individual components that comprise the FLASKDROID architecture.

5.6.2.1 SE Android Module

At the kernel-level, we employ stock SE Android [132] as a building block primarily for the following purposes: First, it is essential for hardening the Linux kernel [175, 174] thereby preventing malicious apps from (easily) escalating their privileges by exploiting vulnerabilities in privileged (system) services. Even when an attack, usually with the intent of gaining *root* user privileges, is successful, SE Android can constrain the file system privileges of the app by restricting the privileges of the root account itself. Second, it complements the policy enforcement at the middleware level by preventing

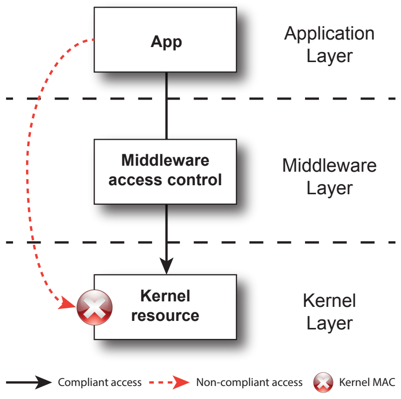

Figure 5.3: SE Android as building block to (1) protect high-value low-level resources and (2) ensure middleware access control mechanisms cannot be bypassed.

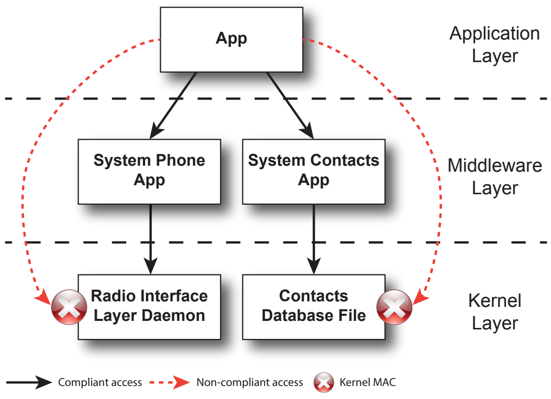

Figure 5.4: Concrete examples for enforcing Android's security model with SE Android.

apps from bypassing the middleware enforcement points (in Flask terminology defined as *User Space Object Managers* (*USOMs*)) and that any privileged operation must go through the Android software stack in a top-down fashion and hence pass all policy enforcement points (cf. Figure 5.3).

Figure 5.4 illustrates two examples for enforcing Android's security model with SE Android and preventing apps from bypassing middleware enforcement points: direct access to low-level system resources such as the radio daemon or the contacts database file can be restricted to be only permissible if coming from the system phone app or system contacts management app. These apps in turn implement access control at middleware level on their public interfaces over which they expose radio or contacts management functionality to other apps.

Dynamic policies. Using the dynamic policy support of SELinux (cf. Section 5.4.1) it is possible to reconfigure the access control rules at runtime depending on the current system state. Our *User Space Security Server* (see following Section 5.6.2.2) is hereby the trusted user space agent that controls the SELinux dynamic policies and can map system states and contexts to SELinux boolean variables (cf. Section 5.6.3). To this end, SE Android provides user space support (in particular *android.os.SELinux*).

Similarly, the SELinux concept of *domain-transitions*—whereby the type of a subject (i.e., process) changes to a new type[5]—can be leveraged to prevent applications from performing low-level operations: For example, an SELinux policy could dictate that the privilege to configure the kernel netfilter at runtime is limited to specific (system) apps. Using the dynamic policy support of SELinux it is possible to reconfigure these access control rules at runtime depending on the current system state. For example, as explained in more detail in Section 5.7.4, a boolean flag could be used to allow 3^{rd}

[5]It is worth noting that *process transition* is a capability that a subject needs to have in order to perform a domain transition. Thus, domain transitions are forbidden unless explicitly allowed by the security policy.

party firewall apps to access the kernel netfilter configuration when the device is not connected to the company network. When connected to the company network, access to the kernel netfilter configuration is restricted so that sensitive network traffic cannot be redirected easily.

5.6.2.2 User Space Security Server

In our architecture, the *User Space Security Server* is the central policy decision point for all user space access control decisions, while the SE Android kernel space security server is responsible for all kernel space policy decisions. This approach provides a clear separation of security issues between the user space and the kernel space components and avoids overloading the kernel security server with access control rules and types that only make sense for middleware entities. Furthermore, it enables at middleware level the use of a more dynamic policy schema (different from the more static SELinux policy language), which takes advantage of the rich semantics (e.g., contextual information) at that layer. Access control is implemented as type enforcement based on (1) the subject type (usually the type associated with the calling app), (2) the object type (e.g., *contacts_email* or the type associated with the callee app UID), (3) the object class (e.g., *contacts_data* or *Intent*), and (4) the operation on the object (e.g. *query*). Since the *User Space Object Managers* (*USOMs*) may be scattered throughout the middleware- and application layers, the *USSS* provides an API for policy checks that is exposed to applications and other system components through the *Context* class every application process on Android uses. However, some middleware *USOMs*, which account for the bulk of the policy checks, e.g., the *ActivityManagerService* and the *PackageManagerService*, are executed inside the system server as well, so no ICC is required to query the *USSS*. The prototypical implementation of our *User Space Security Server* comprises 3,741 lines of Java code.

5.6.2.3 User Space Object Managers

In FlaskDroid, middleware services and apps act as *User Space Object Managers* (*USOMs*) for their respective objects. These services and apps can be distinguished into system components and 3^{rd} party components. The former, i.e., pre-installed services and apps, inevitably have to be *USOMs* to achieve the desired system security and privacy, while the latter can use interfaces provided by the *User Space Security Server* to optionally act as *User Space Object Managers*.

Table 5.1 provides an overview of the *system USOMs* in FlaskDroid and shows some typical operations each object manager controls. Currently, the *USOMs* implemented in FlaskDroid comprise 136 policy enforcement points.

In the following, we explain how we instrument the system *Services* and *Content-Providers* as *User Space Object Managers*. In particular, we highlight the roles of the *PackageManagerService* and *ActivityManagerService* in the design of FlaskDroid.

PackageManagerService. The *PackageManagerService* is responsible for (un-)installation of application packages. Furthermore, it maintains a global list of all available *Activities*, *Services*, *Broadcast Receivers*, and *ContentProviders* contained in the installed

USOM	Example operation
	Service USOMs
PackageManagerService	getPackageInfo getPackageUID findPreferredActivity getInstalledApplications installPackage uninstallPackage
ActivityManagerService	startActivity moveTask grantURIPermission sendBroadcast receiveBroadcast registerBroadcastReceiver
AudioService	adjustStreamVolume getSreamVolume setStreamVolume setRingerMode setVibrateSetting
PowerManagerService	acquireWakeLock isScreenOn reboot preventScreenOn
SensorManager	getSensorList getDefaultSensor unregisterListener registerListener
LocationManagerService	getAllProviders requestLocationUpdates addGpsStatusListener addProximityAlert getLastKnownLocation
ClipboardService	getPrimaryClip setPrimaryClip getPrimaryClipDescription addPrimaryClipChangedListener
SMSManager	copyMessageToIcc deleteMessageFromIcc disableCellBroadcast sendTextMessage
TelephonyManager	getCellLocation getDeviceId listen getNetworkType getCellLocation getMsisdn
AccountManagerService	getAccounts addAccount clearPassword getPassword grantAppPermission
	ContentProvider USOMs
ContactsProvider2	query insert update delete writeAccess readAccess bulkInsert
MMSSMSProvider	query insert update delete
TelephonyProvider	query insert update delete
SettingsProvider	query insert update delete bulkInsert
CalendarProvider2	query insert delete

Table 5.1: List of *system User Space Object Managers* in FlaskDroid with example operations controlled by each manager. Currently, the *USOMs* implemented in FlaskDroid comprise 136 policy enforcement points.

applications in order to find a preferred component for doing a task at runtime. For instance, if an app sends an *Intent* to display a PDF, the *PackageManagerService* looks for a preferred *Activity* able to perform the task.

As a *User Space Object Manager*, we extend the *PackageManagerService* to assign consolidated middleware- and kernel-level app types to all apps during installation using criteria defined in the policy (explained in Section 5.6.3). This is motivated by the fact that at the time a policy is written, one cannot predict which 3^{rd} party apps will be installed in the future. Pre-installed apps are labeled during the phone's boot cycle based on the same criteria. More explicitly, we assign app types to the (shared) UIDs of apps, since (shared) UIDs are the smallest identifiable unit for application sandboxes. In addition, pre-defined UIDs in the system are reserved for particular system components, for instance daemons. Listing 5.1 shows the pre-defined UIDs on Android 4.0.4 as found in *system/core/include/private/android_filesystem_config.h*. With the exception of the system UID (`AID_SYSTEM`) and NFC UID (`AID_NFC`) these UIDs are not assigned to an application package managed by the *PackageManagerService*. Thus, in our implementation we map these UIDs by default to pre-defined types (e.g., `aid_root_t` or `aid_audio_t`) which are hence mandatory types in our system policy.

Furthermore, we extend the logic for finding a preferred component for performing a

Listing 5.1: Pre-defined, reserved UIDs for Android system components

```
AID_ROOT               0  /* traditional unix root user */
AID_SYSTEM          1000  /* system server */

AID_RADIO           1001  /* telephony subsystem, RIL */
AID_BLUETOOTH       1002  /* bluetooth subsystem */
AID_GRAPHICS        1003  /* graphics devices */
AID_INPUT           1004  /* input devices */
AID_AUDIO           1005  /* audio devices */
AID_CAMERA          1006  /* camera devices */
AID_LOG             1007  /* log devices */
AID_COMPASS         1008  /* compass device */
AID_MOUNT           1009  /* mountd socket */
AID_WIFI            1010  /* wifi subsystem */
AID_ADB             1011  /* android debug bridge (adbd) */
AID_INSTALL         1012  /* group for installing packages */
AID_MEDIA           1013  /* mediaserver process */
AID_DHCP            1014  /* dhcp client */
AID_SDCARD_RW       1015  /* external storage write access */
AID_VPN             1016  /* vpn system */
AID_KEYSTORE        1017  /* keystore subsystem */
AID_USB             1018  /* USB devices */
AID_DRM             1019  /* DRM server */
AID_AVAILABLE       1020  /* available for use */
AID_GPS             1021  /* GPS daemon */
AID_UNUSED1         1022  /* deprecated, DO NOT USE */
AID_MEDIA_RW        1023  /* internal media storage write access */
AID_MTP             1024  /* MTP USB driver access */
AID_UNUSED2         1025  /* deprecated, DO NOT USE */
AID_DRMRPC          1026  /* group for drm rpc */
AID_NFC             1027  /* nfc subsystem */

AID_SHELL           2000  /* adb and debug shell user */
AID_CACHE           2001  /* cache access */
AID_DIAG            2002  /* access to diagnostic resources */
```

task at runtime. This implicitly affects the interaction between the system and the user. By default, if several components are suitable to perform a given task (e.g., in the above example several PDF viewers are installed) but no component is set as preferred, the user is prompted to choose one of the potential receivers for performing the task. By filtering this receiver list based on policies, the user-prompt contains only options which would be allowed by the policy, while impossible choices are omitted. The filtering is based on the type of the application that triggers the task, the type of the *Intent* that describes the task, and the type of the potential receiving apps.

Moreover, *PackageManagerService* is responsible for discovering developer provided policies in the application installation packages and forwarding them to the *User Space Security Server*. The *USSS* parses them and considers them during access control decisions that involve the corresponding application (see Section 5.6.2.2).

ActivityManagerService. The *ActivityManagerService* is responsible for managing the stack of *Activities* of different apps, *Activity* life-cycle management, as well as providing the *Intent* broadcast system. As a *USOM*, the *ActivityManagerService* is responsible for labeling *Activity* and *Intent* objects and enforcing access control on them. *Activities* are labeled according to the apps they belong to, i.e., the UID of the application process that created the *Activity*. Subsequently, access control on the *Activity* objects is enforced in the *ActivityStack* subsystem of the *ActivityManagerService*. During operations that manipulate *Activities*, such as moving *Activities* to the foreground/background or destroying them, the *ActivityStack* queries the *USSS* in order to verify that the particular operations are permitted to proceed depending on the subject type (i.e., the calling app) and object type (i.e., the app owning the *Activity* being modified). Listing 5.2 on page 77 presents some examples of these object specific operations for these object classes.

Similar to apps, *Intents* are labeled based on available meta-information, such as the action and category string or the sender app. To apply access control to *Broadcast Intents*, we followed a design pattern as proposed in [140, P4]: We modified the *ActivityManagerService* to filter out receivers which are not allowed to receive *Intents* of the previously assigned type (e.g., to prevent apps of lower security clearance from receiving *Broadcasts* by an app of a higher security clearance). Moreover, the sender of the *Broadcast* can assign a custom label to his *Intent* in order to further classify or also to declassify this *Intent* if in accordance with the policies.

Content Providers. *ContentProviders* are the primary means for apps to share data. This data can be accessed over a well-defined, SQL-like interface. As *User Space Object Managers*, *ContentProviders* are responsible for assigning labels to the data entries they manage during insertion/creation of data and for performing access control on update, query, or deletion of entries. Two approaches for access control are supported: (1) more generic, but rather coarse-grained, based on the URI that identifies particular data entries within a *ContentProvider* or (2) more fine-grained by integrating it into the storage back-end (e.g., SQLite database) for more fine-grained per-data access control.

For approach (2), we implemented a design pattern for SQLite-based *ContentProviders*. Upon insertion or update of entries, we verify that the subject type of the calling app is permitted to perform this operation on the particular object type. To filter queries to the database we create one SQL View[6] for each subject type and redirect the query of each calling app to the respective View for its type. Each View implements a filtering of data based on an access control table managed by the *USSS* which represents the access control matrix for subject/object types. This approach is well-suited for any SQLite-based *ContentProvider* and scales well to multiple stakeholders by using nested Views.

Services. A *Service* is a component of an application that provides a particular functionality to other (possibly remote) components, which access the *Service* component via ICC. To this end, *Service* interfaces are exposed as Binder IPC objects that are

[6]A virtual SQL table, which represents a subset of the original table's content based on an inherent select statement.

generated based on an interface specification described in the Android Interface Definition Language (*AIDL*).

To instrument a *Service* as a *User Space Object Manager*, we add access control checks when a (remote) component connects to the *Service* and on each call to *Service* functions exposed by the *Service* API. Since those checks are added to each *Service* implementation and not to a central component, we want to automate this instrumentation as far as possible. We approached this scalability problem by extending the lexer and parser of Android's *AIDL* compiler to recognize (developer-defined) type tags on *Service* interface and function declarations. The *AIDL* code generator was extended to automatically insert policy checks for these types in the auto-generated *Service* stub code. Thus, the scalability problem is reduced to setting the types of the service and of its functions by adding type-tags to their definitions. Moreover, since both system *Services* (with very few exceptions) and 3^{rd} party app *Services* are defined in *AIDL* and generated with the *AIDL* compiler, our extension instruments both system *Services* and 3^{rd} party app *Services* in the same way.

5.6.2.4 Context Providers

A context is an abstract term that represents the current security requirements of the device. It can be derived from different criteria, such as physical criteria (e.g., the location of the device) or the state of apps and the system (e.g., the app being currently shown on the screen). To allow for flexible control of contexts and their definitions, our design employs *Context Providers*. These providers come in form of plugins to our *User Space Security Server* (see Figure 5.2) and can be arbitrarily complex (e.g., use machine learning) and leverage available information such as the network state or geolocation of the device to determine which contexts apply. *Context Providers* are notified by the system about context changes similar to the approach taken in [24]. Each *Context Provider* is responsible for a distinct set of contexts, which it (de-)activates in the *USSS*. Decoupling the context monitoring and definition from our policy provides that context definitions do not affect our policy language except for very simple declarations (as we will show in Section 5.6.3.1).

Moreover, the *USSS* provides feedback to *Context Providers* about the performed access control decisions. This is particularly useful when instantiating security models like [P5, 24] in which access control decisions depend on the history of decisions. A good example for the usefulness of such a feedback channel is an architecture like XManDroid [P5], which aims at mitigating collusion attacks between different apps. In XManDroid, the set of allowed IPC channels of an app is directly dependent on its past IPC behavior. This approach can be instantiated with FlaskDroid with a plugin that models the currently established IPC channels (e.g., as a graph [P5]) based on the granted access control decision. The plugin internally evaluates this graph and sets the contexts such that each application's IPC channels are restrained such that no collusion attack is feasible.

To support context-aware policies in our FlaskDroid prototype, we implement different *Context Providers*, which register Listener threads to be notified about context changes similar to the approach taken in [24]. In our current implementation, the context is derived from the location information provided by the GPS sensor, the user presence detected by the *ActivityManagerService*, the *ActivityStack*, and a notification *Intent* by the telephony app. These information are matched against defined contexts within the *Context Providers*.

5.6.3 Policy

In this section we explain the policy language used in FlaskDroid to express the access control rules.

5.6.3.1 Policy Language and Extensions

We extend SELinux's policy semantics for *type enforcement* (cf. Section 5.4) with new default classes and constructs for expressing policies on both middleware and kernel-level. A recapitulation of the SELinux policy language is out of scope of this dissertation and we focus here on our extensions.

New default classes. Objects are distinguished by their *class*, e.g., file or socket, and *operations* the object class supports, e.g., read, write, open. Similar to classes at the kernel-level, like *file* or *socket*, we introduce new default classes and their corresponding operations to represent common objects at middleware level, such as *Activity*, *Service*, *ContentProvider*, *Broadcast*, and *Intent*. Operations for these classes are, for example, *startActivity*, *query* a *ContentProvider*, or *receive* a *Broadcast*. Listing 5.2 presents examples for the definition of basic classes, such as *activity_c* or *service_c* and illustrates class inheritance at the example of *intentService_c*, which inherits from *service_c*, as well as several *ContentProviders* like *contactsProvider_c*, which inherit from the *contentProvider_c* class.

Listing 5.2: Example policy snippet illustrating the definition of middleware-specific object classes including inheritance between classes.

```

class activity_c
{
    start stop grantURIPermission finish moveTask
};

class service_c
{
    start stop bind callFunction find
};

class intentService_c inherits service_c
{
    onHandleIntent
};

class clipBoardService_c
{
    getPrimaryClip
};

```

```
class broadcast_c
{
    send receive sendSticky receiveSticky registerReceiver ↷
       unregisterReceiver
};

class intent_c
{
    send receive
};

class contentProvider_c
{
    query insert update delete readAccess writeAccess
};

class contactsProvider_c inherits contentProvider_c;
class calendarProvider_c inherits contentProvider_c;
class downloadProvider_c inherits contentProvider_c;
class mediaProvider_c inherits contentProvider_c;
class mmssmsProvider_c inherits contentProvider_c;
class smsProvider_c inherits contentProvider_c;
class telephonyProvider_c inherits contentProvider_c;
class settingsProvider_c inherits contentProvider_c;

class app_c
{
    clearAppUserData checkPermission switch
};

class package_c
{
    getPackageInfo getPackageInfoWithUninstalled
    getPackageUID getPackageGIDs getPackagesForUid
    getNameForUid getUidForSharedUser
    findPreferredActivity queryIntentActivities
    getInstalledApplications
    getInstalledApplicationsWithUninstalled
    getInstalledPackages
    getInstalledPackagesWithUninstalled
};

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

class sensorService_c
{
  getSensorList getDefaultSensor unregisterListener registerListener
};
```

Application Types. A further extension is the possibility to define criteria by which applications are labeled with a security type. The criteria for apps can be, for instance, the application package name, the requested permissions or the developer signature.

Listing 5.3 illustrates an example policy snippet defining the criteria for assigning a type to applications (i.e., labeling apps). The criteria at middleware level can be, for instance, the application package name, the requested permissions, the developer signature, or potential, non-standard meta-information such as an external signature (lines 9–48). To illustrate this more concretely, consider a newly installed application whose package name equals *com.android.apps.tag*. According to the lines 29–32 in the policy in Listing 5.3, this app would be assigned the type *app_tag_t*, since this the criteria for type *app_tag_t* match the new app. If no criteria matched a specific app, a default type is assigned (line 4). In FLASKDROID, labels are assigned to the sandbox of applications, i.e., to their UID or in case of shared UIDs to their shared UID. The latter decision is motivated by the fact that UID is the smallest identifiable unit provided by Binder to the *USOMs*.

Listing 5.3: Example policy snippet illustrating the definition of criteria for assigning types to apps.

```
 1 /*
 2 Default type
 3 */
 4 defaultAppType untrustedApp_t;
 5
 6 /*
 7 Define criteria to assign types to apps
 8 */
 9 appType app_cased_t
10 {
11     Developer:signature=0xFEF9...;
12     Package:package_name=de.cased.trust.app;
13     ExternalSignature:keyFileLocation=/etc/key.file;
14     ExternalSignature:signatureFileLocation=assets/sig.file;
15 };
16
17 appType android_t
18 {
19   /* All of packages under this UID */
20   Package:package_name=android;
21   Package:package_name=com.android.keychain;
22   Package:package_name=com.android.settings;
23   Package:package_name=com.android.seandroid_manager;
24   Package:package_name=com.android.providers.settings;
25   Package:package_name=com.android.systemui;
26   Package:package_name=com.android.vpndialogs;
27 };
28
29 appType app_tag_t
```

```
{
  Package:package_name=com.android.apps.tag;
};

appType app_backupconfirm_t
{
  Package:package_name=com.android.backupconfirm;
};

appType app_telephony_t
{
  Package:package_name=com.android.phone;
  Package:package_name=com.android.providers.telephony;
};

appType app_bluetooth_t
{
  Package:package_name=com.android.bluetooth;
};

[...]
```

Intent types. Similarly to apps, also *Intents* may be the object of access control enforcement, e.g., if a particular app is allowed to send or receive an *Intent* of a particular type. Thus, also for *Intents* we require criteria to label *Intent* objects. Criteria for assigning a type to *Intent* objects can be the *Intent* action string, category or receiving component. Listing 5.4 illustrates the definition of such criteria for assigning a type to *Intent* objects. It leverages the information available about *Intents* such as their action string or category, or receiving component (lines 6-11). Again, if no criteria matched an *Intent*, a default type is used (line 4). For instance, the example in Listing 5.4 would assign an *Intent* with action string *android.intent.action.MAIN*, category *android.intent.category.HOME*, and type *app_launcher_t* of the receiving component the type *intentLaunchHome_t*. In contrast to apps, which are labeled once during installation, *Intents* are labeled on-demand at runtime during policy checks that involve *Intent* objects and, naturally, the assigned type is bound to the life-time of the corresponding *Intent* object.

Listing 5.4: Policy snippet showing definition of criteria for assigning an Intent a specific type.

```
 1 /*
 2 Default type
 3 */
 4 defaultIntentType untrustedIntent_t;
 5
 6 intentType intentLaunchHome_t
 7 {
 8     Action:action_string=android.intent.action.MAIN;
 9     Categories:category=android.intent.category.HOME;
10     Components:receiver_type=app_launcher_t;
11 };
```

Context definitions and awareness. We extend the policy language with an option to declare *contexts* to enable context-aware policies. Listing 5.5 shows the declaration of three contexts, `work_con`, `phoneBooth_con`, and `iptablesExecForbidden_con`. Each declared context can be either *actived* or *deactived* by a dedicated *Context Provider*.

Listing 5.5: Policy snippet showing declaration of contexts.

```
context work_con;
context phoneBooth_con;
context iptablesExecForbidden_con;
[...]
```

To actually enable context-aware policies, we introduce in our policy language *switchBoolean* statements that map contexts to booleans, which in turn provide dynamic policies. These booleans exist only at the middleware and affect only the policy decision making in the *USSS*. To map contexts to the kernel-level, we introduce *kbool* definitions, which point to a boolean at kernel level instead of adding a new boolean at middleware. Changes to such kernel-mapped boolean values by *switchBoolean* statements trigger a call to the SELinux kernel module to update the corresponding SELinux boolean. On a context switch reported by the system, all boolean variables that relate to the new context are set to their respective values. This enables or disables the policy rules that depend on those boolean values. When a context is unset, all related booleans can be optionally auto-reverted to their original value or, alternatively, one can define other contexts that trigger a change of those booleans.

To enable runtime configuration of the kernel MAC booleans, SE Android provides user space support (in particular *android.os.SELinux*) and our middleware extensions take the role of a trusted user space agent which manages SE Android's booleans. Self-contained system and kernel MAC policies ensure that only the system server, which executes our extensions, is allowed to use this mechanism.

Listing 5.6 presents the definition of a boolean *phoneBooth_b* at middleware level (line 4) and references to a boolean *allowIPTablesExec_b* defined in the underlying SE Android policy (line 9). Both, contexts (cf. Listing 5.5) and booleans, are used in *switchBoolean* statements (lines 15-27), which define which booleans are (un-)set depending on which context is (in-)active. To illustrate this, consider the *switchBoolean* statement in lines 15-20 in Listing 5.6, which defines that as soon as the context *phoneBooth_con* is active, the middleware boolean *phoneBooth_b* has to be set to *true*. As soon as the *phoneBooth_con* context is deactivated, the *phoneBooth_b* boolean should be reset to its original value, i.e., *False* (line 4). The *switchBoolean* in lines 22-27 works identical, however, for the context *iptablesExecForbidden* and the boolean *allowIPTablesExec_b*, which is an SE Android boolean and hence the change in value is mapped to the kernel-level.

Listing 5.6: Policy snippet showing how booleans are linked with contexts.

```
1 /*
2 Middleware boolean definitions
3 */
4 bool phoneBooth_b = false;
5
```

```
/*
Kernel boolean defintion used for sync with SE Android
*/
kbool allowIPTablesExec_b = true;

/*
Dynamic policies
*/

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

switchBoolean
{
    context_id=iptablesExecForbidden_con;
    auto_reverse=true;
    allowIPTablesExec_b=false;
}

[...]
```

Policy rules. Listing 5.7 shows the definition of some example access control rules. To ease writing policies, all parameters can be *sets* of logically disjunct parameters, i.e., the parameter matches if any of the set members matches. For instance, the rule in line 5 of Listing 5.7 has as *subject_type* parameter the set of types *{app_system_t app_contacts_t app_launcher_t}*, as *object_type* the type *allContactsData_t*, as *object_class* parameter the class *contactsProvider_c*, and as *operation* the function *query*. Optionally, these rules can also depend on boolean parameters that enable or disable policy rules. This dependency is noted in form of *if* statements in the rules. For instance, the two rules in lines 17 and 18 in Listing 5.7 depend on the boolean *phoneBooth_b* and are only valid if *phoneBooth_b* is *True* or are invalid if is *False*, respectively.

To further ease writing access control policies, we adopt SELinux' *self* keyword, meaning that this rule applies always when subject type and object type are identical. Lines 1-3 of Listing 5.7 show examples of such rules. For instance, the rule in line 3 states, that any app can send and receive broadcasts to and from apps with the identical type.

Additionally, we add the keyword *any*, which can be used as a wildcard for the subject type, object type, object class, and operations and matches any argument.

Listing 5.7: Policy snippet showing definition of access control rules (optionally depending on boolean parameters).

```
1 self: app_c {checkPermission};
2 self: activity_c {finish moveTask};
3 self: broadcast_c {receive send};
4
```

```
allow {app_system_t app_contacts_t app_launcher_t} allContactsData_t: ↪
    contactsProvider_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};
};

[...]
```

5.6.3.2 Support for Multiple Stakeholders

A particular requirement for the design of FlaskDroid is the protection of interests of different stakeholders. This requires, that policy decisions consider the policies of all involved stakeholders. These policies can be pre-installed (i.e., system policy or enterprise policy), delivered with apps (i.e., app developer policies), or configured by the user (e.g., *User Policy App* in Figure 5.2). The *PackageManagerService* is instrumented such that it extracts policy files included in APKs during app installation and injects them into the *USSS*. Thus, in FlaskDroid 3^{rd} party developers have the choice to deploy custom policies for their applications. Note, that this is completely voluntary on their part and they may choose to opt in and rely on our security framework to enforce their policies. In case they opt out from using our framework (or they are unaware of it), their applications are still subject to the system and user policy enforcement.

Moreover, 3^{rd} party app developers can choose to instrument their app components as *User Space Object Managers* for their own data objects. FlaskDroid provides the necessary interfaces to query the *User Space Security Server* for policy decisions as part of the SDK (i.e., *Context* class mentioned in Section 5.6.2.2). These decisions are based on the app-specific 3^{rd} party policy, which defines custom *appType* statements to label subjects (e.g., other apps) and declares app-specific object types. To register app-specific policies, the *PackageManagerService* is instrumented such that it extracts policy files during app installation and injects them into the *USSS*.

To differentiate between the different policies, we leverage namespaces. In each namespace, security types and objects classes can be defined that are independent of the other namespaces. Thus, for instance, app developers can define app-specific policies with custom app and *Intent* types that are only valid within their own namespace.

A particular challenge when supporting multiple stakeholders is the reconciliation of the various stakeholders' policies. Different strategies for reconciliation are possible [152, 120] and generally supported by our architecture, based on namespaces and global/local type definitions. For instance, as discussed in [152], *all-allow* (i.e., all stakeholder policies must allow access), *any-allow* (i.e., only one stakeholder policy must allow access), *priority* (i.e., higher ranked stakeholder policies override lower ranked ones), or *consensus* (i.e., at least one stakeholder policy allows and none denies or vice versa). However, choosing the right strategy strongly depends on the use-case. For example, on a pure business smartphone without a user-private domain, the system (i.e., company) policy always has the highest priority, while on a private device a consensus strategy may be preferable.

In our implementation we opted for a consensus approach, in which the *system* policy check is mandatory and must always consent for an operation to succeed.

5.6.3.3 Policy Implementation

Listing 5.8 presents the full grammar of our policy language as implemented and used throughout the policy listings in this chapter. The grammar is noted in *Extended Backus-Naur Form*. We implemented a Python-based compiler for this language, using the *pyparsing* and *ElementTree* libraries, which produces an XML representation of the policy that is used by our middleware extensions. For sanity checks, the compiler tool verifies the XML output against an XML Schema of the policy language as supported by our extensions on the device.

Listing 5.8: Grammar of our policy language in Extended Backus Naur Form.

```
Policy ::= DefaultDecision, DefaultAppType, DefaultIntentType, ↷
    PolicyElements ;

(* Default definitions *)
DefaultDecision ::= "defaultDecision", DecisionValue ;
DefaultAppType ::= "defaultAppType", Identifier ;
DefaultIntentType ::= "defaultIntentType", Identifier ;

(* Policy construct *)
PolicyElements ::= {Attribute}, {Type}, {Boolean}, {KBoolean}, {Class},↷
     {Rulestatement}, {Context}, {SwitchBoolean}, {AppType}, {↷
    IntentType} ;

(* Policy element definitions *)
Attribute ::= "attribute", Identifier, ";" ;
Type ::= "type", #(Identifier), ";" ;
Boolean ::= "boolean", Identifier, "=", BooleanValue, ";" ;
```

```
KBoolean ::= "kboolean", Identifier, "=", BooleanValue, ";" ;
Class ::= "class", Identifier, [ "inherits", Identifier ], [ ↷
    BraceIdentifierlist ] ";" ;
Rulestatement ::= "if", "(", Identifier, ")", "{", 1*(Rule), "}", ";" |↷
     1*(Rule) ;
Rule ::= "allow", SubjectType, ObjectType, ":", ObjectClass, Permission↷
     ";"
     | "self", ":", ObjectClass, Permission ";" ;
Context ::= "context", Identifier, ";" ;
SwitchBoolean ::= "switchBoolean", sbBody, ";" ;
sbBody ::= "{", contextAssignment, autoReverse, 1*(BoolAssignment), "}"↷
     ;
contextAssignment ::= "context_id", "=", Identifier, ";" ;
autoReverse ::= "auto_reverse", "=", BooleanValue, ";" ;
AppType ::= "appType", Identifier, DomainAssignmentlist, ";" ;
IntentType ::= "intentType", Identifier, DomainAssignmentlist, ";" ;

(* Basic constructs *)
KVAssignment ::= Identifier, "=", Value, ";" ;
BoolAssignment ::= Identifier, "=", BooleanValue, ";" ;
Assignmentlist ::= "{", 1*(KVAssignment), "}" ;
DomainAssignmentlist ::= "{", 1*(Identifier, ":", KVAssignment), "}" ;
Identifierlist ::= Identifier, *(whitespace, Identifier) ;
BracketIdentifierlist ::= "[", Identifierlist, "]" ;
BraceIdentifierlist ::= "{", Identifierlist, "}" ;
SubjectType ::= (Identifier | BraceIdentifierlist) ;
ObjectType ::= (Identifier | BraceIdentifierlist) ;
ObjectClass ::= (Identifier | BraceIdentifierlist) ;
Permission ::= BraceIdentifierlist ;

(* Basic definitions *)
DecisionValue ::= "deny" | "allow" ;
BooleanValue ::= "true" | "false" ;
Identifier ::= 1*(alphanum | "_") | "*";
Value ::= [ "~" ], 1*(alphanum | "_" | ":" | "." | "/") ;

(* Basic terminals *)
alphanum ::= { alpha | digit } ;
alpha ::= "A" | ... | "Z" | "a" | ... | "z" ;
digit ::= "0" | ... | "9" ;
whitespace ::= ? white space characters ? ;
```

5.7 Use-Cases / Instantiations

As mentioned in our threat model (cf. Section 5.5.1), the recent incidents on smartphone privacy and security breaches have led to a demand for privacy protecting as well as (enterprise) security solutions in practice. In the following we will introduce some of them and show how FlaskDroid can instantiate them. Moreover, we will discuss how FlaskDroid can go beyond these specific solutions, for instance, by securely integrating virus scanner or firewall management apps into the overall system.

5.7.1 Privacy Enhanced System Services and Content Providers

System *Services* and *ContentProviders* are an integral part of the Android application framework and implement the API exposed to 3^{rd} party apps through the Android SDK. Prominent services are, for instance, the *LocationManager* or the *Audio Service* and prominent *ContentProviders* are the contacts app and SMS/MMS app. By default, Android enforces permission checks on access to the interfaces of these *Services* and *ContentProviders*.

Problem description: However, it is known that the default permissions are too coarse-grained and protect only access to the entire *Service*/*Provider* but not to specific functions or data. Moreover, they are static and cannot be selectively revoked (prior to the upcoming Android M release). Thus, the user cannot control in a fine-grained fashion, which sensitive data can be accessed how, when and by whom. For instance, the popular WhatsApp service has in the past been shown to upload user's contacts data [202]. Also other apps such as Facebook have access to the entire contacts database although only a subset of the data (i.e., names and email addresses) is required for their correct functioning. On the other hand, recent attacks demonstrate how even presumably privacy-unrelated and thus unprotected data such as the accelerometer readings can be misused against user's security and privacy [211, 19].

Solution: We tagged selected query functions of the system *AudioService*, *LocationManager*, and *SensorManager* with specific security contexts (e.g., *object_type* as *fineGrainedLocation_t*, *object_class* as *locationService_c*, and *operation* as *getLastKnownLocation*) and used our extended *AIDL* compiler (see Section 5.6.2.3) to auto-generate fine-grained authorization hooks for those services. This, particularly in conjunction with support for context-aware enforcement, enables privacy enhancing policies. For instance, calling functions with *Service* object types is prohibited while the phone is in a security sensitive state. Thus, retrieving accelerometer information or recording audio would not be possible when, e.g., the virtual keyboard/PIN pad is in the foreground or a phone call is in progress (both definable phone states).

In Section 5.6.2.3 we also explained how *ContentProviders* (e.g. the *ContactsProvider*) can be instrumented as *User Space Object Managers*. With system *ContentProvider USOMs*, users can refine the system policy rules for different private data, such as their contacts data. A user can, for instance, grant a social-networking app read access to their "friends" and "family" contacts' email addresses and names, while prohibiting it from reading their postal addresses and any data of other groups such as "work." For technical details on how this is implemented and on the policies, we refer to our technical report [T2] on privacy enhanced *ContentProviders*.

5.7.2 Multi-level Security

Smartphones are increasingly applied in scenarios where different security domains are desired. The most prevalent example today are corporate smartphones which are simultaneously used for private purposes or vice versa (i.e., based on *"Bring your own device"* philosophy). Previous work [P4] has acknowledged this need for domain isolation on dual used smartphones and our FLASKDROID architecture can efficiently and effectively instantiate this use-case.

Problem description: Android does not provide support for declaring different security domains and strongly isolating them from each other. Thus, malicious apps in or attacks against the private domain on the phone can compromise the corporate domain.

Solution: Our architecture supports multi-level security (MLS) and thus with FlaskDroid business applications can be clearly distinguished and labeled with a corresponding type during installation (cf. Section 5.6). At runtime, the non-business and business domain are securely isolated from each other at middleware level and kernel level. For instance, only apps from the business domain are permitted to contact the *Services* of another business domain app. Moreover, contacts created for the business domain (i.e., of group "work") could only be read and written by application from the business domain (see also the use-case in Section 5.7.1). Similarly, our consolidated kernel- and user space policies can ensure that despite DAC, files created by a business app could only be accessed by other business apps.

5.7.3 Secure Logs

A further use-case of our interest concerns the log facility of Android. Android applications can write entries to the *log* facility by either writing directly to the world-writeable */dev/log/** device files, by using the *log* and *logwrapper* tools or the Android Log API.

Problem description: Android apps with an API version smaller than 16 that hold the *READ_LOGS* permission effectively become member of the log group that has read access to the kernel log facilities (*/dev/log/**; enforced by the Linux DAC, similar to the Internet permission). On devices with API version higher than 16 (introduced with Android version 4.1), this permission is only available to system apps.[7] Applications with access to the logs use by default the *logcat* tool available on the phone to read and parse the entries in the */dev/log/** sysfs files.[8] However, the capability to read the logs has been shown to be a security and privacy threat. For instance, sensitive information are frequently logged by apps and thus could leak via the log [111, 145].

Solution: We use SELinux to enforce that only the Android *logcat* tool is allowed to read from */dev/log/** using a domain transition when *logcat* is executed. With *logcat* now being the only access point to log entries, it can be extended as a *User Space Object Manager*: Upon read access, the log facility filters all entries from the result, for which the reading process type does not have the required security clearance. Policy queries from the logcat tool to our middleware *USSS* are performed via a dedicated socket provided by the *USSS*. This design strongly resembles the exemplary use-case for the SELinux access control on the */etc/shadow* file on Linux systems. Processes can only modify this security sensitive file through the *passwd* program, which is inherently trusted to enforce that calling processes can only modify entries of their own user. However, in contrast to this default use-case, our secure logs solution requires the joint operation of both user space and kernel space, where the log facility of the kernel is responsible for object labeling (i.e., log entries) and the logcat tool in user space is

[7] `https://code.google.com/p/android/issues/detail?id=34785`

[8] While it is possible to read */dev/log/** directly, in practice, all applications we analyzed use the logcat tool.

responsible for type enforcement (i.e., on read). Listing 5.9 presents a simple policy for enforcing access control on log entries. In this case, every application has read access only to its own logging information in the default log, but no other (line 1). Only apps from the system application domain are allowed to read log entries from both the default log (`log_c`) and the system log (`syslog_c`) independently of the log entries' security type (line 2).

Listing 5.9: Example policy snippet illustrating implementing different security types on log entries.

```
1 self: log_c { read };
2 allow systemApp_a any: { log_c syslog_c } { read };
```

5.7.4 Firewall and Anti-Virus Apps

In default Android, certain permissions like Internet or Bluetooth are mapped to Linux groups (cf. Background Section 2.2 on Android's security architecture). Similarly, our architecture is used to introduce new capabilities, which, in contrast to permissions, can be revoked at runtime using booleans. As example, we introduced the capabilities that a 3^{rd} party app can act as a manager for the Linux kernel packet filter, i.e., it can execute the *iptables* tool with sufficient privileges, and the capability to inspect other apps' APK packages (both, the public and the private portion, cf. *forward locking*) in order to perform anti-virus scans. While this could be achieved by adding new permissions to the system, using FlaskDroid is much more flexible and allows for efficient upgrades of the policy as well as more fine-grained access control.

Problem description: Default Android does not support this use-case and requires the user to root his phone to enable contemporary available firewall and anti-virus apps with the above described functionality. Naturally, this severely impedes the platform security and opens the door for other malware that leverages the root privileges.

Solution: We install the *iptables* binary with setuid bit and user root. Additionally, we assign this binary an SELinux security label with *fwapp* as role, so that only apps that have been assigned this role (or that inherited this role) can execute the binary. Similarly, we introduce a role *avapp* which is allowed to inspect the private files of other apps.[9] These SELinux roles are assigned to apps during installation (i.e., to their UID) and in future work we plan to extend this install process such that apps can requests roles in their manifest and depending on user consensus or their app type (cf. Section 5.6.3) the role is granted or denied to the application.

5.7.5 Phone Booth Mode

Users may lend their mobile device to an acquaintance (or even stranger) to make a phone call. The goal of this use-case is to provide the user with the means to do so

[9]This includes, that the DAC permissions are relaxed, since the access must be allowed by DAC **and** MAC

securely and without the need to worry about his private information stored on his device. Thus, the privacy objective is to lock down the device in such a way that the acquaintance can still make a phone call, but has no privileges on the phone beyond this (e.g., reading call logs or the address book, or even open other apps than the dialer app).

Problem description: The user loses or gives up physical control over his device. This can pose a serious threat to the user's privacy, since the other person could freely inspect the phone data and apps. For instance, the user's address book and call history are immediately visible within the dialer app, and other apps usually do not apply further authentication mechanisms that would stop someone with physical access from opening and inspecting them.

Solution: With Android version 4.3, Google introduced the *restricted profiles* feature [70] that allows the device owner to define user profiles with configurable access to installed apps. With Android version 5.0 a dedicated, unprivileged *guest mode* [76] was added and, further, the *screen pinning* [71] that allows pinning a particular app to the top of the *ActivityStack* and, hence, preventing the user from (accidentally) closing it. However, prior to version 4.3 none of these features were available and this FlaskDroid use-case, which targeted Android version 4.0.4, was thus clearly ahead of its time. In this use-case, we used FlaskDroid's fine-grained access control within the *ActivityManagerService* and *ActivityStack* to control if another application can be moved into or off the foreground on the screen. This resembles the later introduced screen pinning feature, however, in a FlaskDroid policy-driven way: The user can trigger a transition into a special phone-context "phone booth mode" by pressing a button, in which the policy dictates that the dialer app is the only app permitted to be in the foreground. Simultaneously, the policy defines that all access to the *CallLogProvider* and *ContactsProvider* must be denied, meaning they return empty results to all queries, thus preventing the phone app from auto-completing entered telephone numbers or from showing the call log/contacts data. In this mode, the user can safely lend his phone to another person for the sake of making a phone call.

To exit this mode, the phone app informs the *User Space Security Server* about the changed state and thus resets the corresponding boolean values that influence access to *ActivityStack* and *CallLogProvider*/*ContactsProvider* to their original state. This context change is triggered using a special combination of the default soft keys that are part of the trusted UI region (i.e., are not accessible to or exchangeable by 3^{rd} party apps). To authenticate the context change, different options exist: For instance, the user is prompted to enter his PIN (as in the Android screen pinning scenario), configured password, or on more recent hardware using biometrics.

Although this use-case focused on the dialer application, the same technique could be used to provide a generic "kiosk mode" in the same way the (now) default screen pinning feature in conjunction with restricted profiles works.

For the example policy that implements this use-case, we refer to our publication [P3] or technical report [T3].

5.7.6 App Developer Policies (Saint)

Ongtang et al. present in [140] an access control framework, called *Saint*, that allows app developers to ship their apps with policies that regulate access to their app's components.

Problem description: The concrete example used to illustrate this mechanism consists of a shopping app whose developer wants to restrict the interaction with other 3^{rd} party apps to only specific payment, password vault, or service apps. For instance, the developer specifies that that the password vault app must be at least version 1.2 or that a personal ledger app must not hold the Internet permission. Saint's policy uses the source app plus an optional Intent object, the destination app, some conjunctional conditions (e.g., permissions or signature key of the destination app), as well as the system state (e.g., physical location or bluetooth state) as parameters for access control rules.

Solution: Instantiating Saint's runtime access control on FlaskDroid is achieved by mapping Saint's access control parameters to the ones supported by FlaskDroid: the source app, the optional Intent object, the destination app, and the conditions can be combined into security types for the subject (i.e., source app) and object (i.e., destination app or Intent object). For instance, a specific type is assigned to an application with a particular signature and permission. If this app is source in the Saint policy, it is used as *subject_type* in FlaskDroid policy rules; and if it is used as destination, it is used as *object_type*. The *object_class* and *operation* are directly derived from the destination app. The *system state* can be directly represented by FlaskDroid policy.

Listing 5.10 shows an instantiation of the developer policy in [140] on our architecture. The depicted policy is deployed by the shopping app and thus `self_t` refers to the shopping app. We define types `app_trustedPayApp_t`, `app_trustedPayApp_t`, `app_noInternetPerm_t` (lines 1-3 and lines 8-22) for the specific apps with which the shopping app is allowed to interact and describe some of the allowed interactions by means of *Intent* types `intent_actionPay_t` and `intent_recordExpense_t` (lines 5-6 and lines 24-28). Afterwards, we declare access control rules that reflect the desired policy described in the example in [140] (lines 36-38). For instance, the rule in line 36 defines that the shopping app is allowed to send an *Intent* with action string `ACTION_PAY` only to an app with type `app_trustedPayApp_t` (line 27), which in turn is only assigned to apps with the developer signature `308201...` (line 10). The rule in line 37, on the other hand, defines that the shopping app is allowed to interact in any way with an app with type `app_trustedPWVault`, which is only assigned to apps with package name `com.secure.passwordvault` and minimum version `1.2` (lines 15-16). Naturally, the developers of the other apps (e.g., of `com.secure.passwordvault` or with signature `308201...`) have to ship corresponding policies to concur to this interactions (e.g., receiving *Intent* from the shopping app).

Listing 5.10: Policy snippet showing instantiation of Saint (140) example for runtime policy enforcement. Policy snippet is from the policy deployed by the shoping app.

```
1 type app_trustedPayApp_t;
2 type app_trustedPWVault_t;
3 type app_noInternetPerm_t;
```

```

type intent_actionPay_t;
type intent_recordExpense_t;

appType app_trustedPayApp_t
{
    Developer:signature=308201...;
};

appType app_trustedPWVault_t
{
    Package:package_name=com.secure.passwordvault;
    Package:min_version=1.2;
};

appType app_noInternetPerm_t
{
    Package:permission=~android.permission.INTERNET;
};

intentType intent_actionPay_t
{
    Action:action_string=ACTION_PAY;
    Components:receiver_type=app_trustedPayApp_t;
};

intentType intent_recordExpense_t
{
    Action:action_string=RECORD_EXPENSE;
    Components:receiver_type=app_noInternetPerm_t;
};

allow self_t intent_actionPay_t: intent_c { send };
allow self_t app_trustedPWVault_t: any { any };
allow self_t intent_recordExpense_t: intent_c { send };
```

Although both the original Saint policy and its instantiation on our solution achieve the same security objectives, the policy languages between the two systems differ. Most noteworthy is, that in Saint's policy language the subjects and objects are defined within the access control rules. Thus, if two rules use identical subjects and objects, the definition of those is redundantly repeated. Moreover, in Saint it is not possible to group several distinct subjects or objects into sets and thus ease writing of rules which differ only in the object class or operation. In FLASKDROID, the policy language requires the policy author to first clearly define all possible types (for subjects and objects) as well as object classes and their respective operations. While this might seem at first glance more tedious, it greatly eases authoring of policy rules afterwards since one can use these types without redundantly repeating their definition and, moreover, can group types, classes, or operations into sets and thus achieve more compact and readable rules.

5.8 Evaluation

In this section we evaluate and discuss our architecture in terms of policy design, effectiveness, and performance overhead.

5.8.1 Policy

To evaluate our FLASKDROID architecture, we derived a basic policy that covers the pre-installed system *USOMs* that we introduced in Section 5.6.2.3.

5.8.1.1 Policy Assessment.

Security policies, including access control policies, are generally evaluated to be *good* based on different properties, such as *safety* [89], *completeness* [2, 97], and *effectiveness.* The development of a security policy that fulfills these properties is a highly complex process. For instance, on SELinux enabled systems the policies were incrementally developed and improved after the SELinux module had been introduced, even inducing research on verification of these properties [100, 97]. A similar development can be currently observed for the SE Android policies which are written from scratch [175]. For FLASKDROID we are for now foremost interested in generating a basic policy to estimate the access control complexity that is inherent to our design, i.e., the number of new types, classes, and rules required for the *system User Space Object Managers*, and hence lay the foundation for the development of a *good* policy.

5.8.1.2 Policy Generation

Established approaches. Different approaches exist to generate access control policies. For instance, manual authoring of very special purpose and use-case specific policies as we have shown in Section 5.7. For more complex policies, (semi-)automatic methods have been proposed. For example, the *polgen* tool for SELinux [178] processes the traces of the dynamic behavior of target process (e.g., information flow patterns) and generates new types and policy rules. *polgen* operates human-aided and semi-automated, since a human has to determine the exact security policies and adjust the generated policies via wizard-style interface. Similarly, the benefits of using the system call traces for guided generation of a policy for fine-grained process confinement have been shown [150]. Also static analysis of target binaries seems a feasible approach to help automating policy generation and has been already employed to verify the information flow properties of SELinux policies [167].

Policy generation for our evaluation. To generate our basic policy, we opted for an approach that follows the concepts of TOMOYO Linux' learning phase[10] and other semi-automatic methods [150]. The underlying idea is to derive policy rules directly from observed application behavior. To generate a log of system application behavior, we leveraged FLASKDROID's audit mode, where policy checks are logged but not enforced.

[10]http://tomoyo.sourceforge.jp/2.2/learning.html.en

Under the assumption, that the system contained in this auditing phase only trusted apps, this trace can be used to derive policy rules.

To achieve a high coverage of app functionality and thus log all required access rights, we opted for testing with human user trials for the following reasons: First, automated testing has been shown to exhibit a potentially very low code coverage [65, 66] and, second, Android's extremely event-driven and concurrent execution model complicates static analysis of the Android system [215, 66].

The users' task was to thoroughly use the pre-installed system apps by performing various every-day tasks (e.g., maintaining contacts, writing SMS, browsing the Internet, or using location-based services). To analyze interaction between apps, a particular focus of the user tasks was to leverage inter-app functionality like sharing data (e.g., copying notes from a website into an SMS). For testing, the users were handed out Galaxy Nexus devices running FlaskDroid with a *No-allow-rule* policy. This is a manually crafted policy containing only the required subject/object types, classes and operations for the *USOMs* in our architecture, but no allow rules. The devices were also pre-configured with test accounts (e.g., EMail) and test data (e.g., fake contacts).

Derived basic policy. Using the logged access control checks from these trials, we derived 109 access control rules required for the correct operation of the system components (as observed during testing). Table 5.2 provides an overview of the basic policy derived from all user test results and the additionally deployed SE Android policy (above the double line in the Figure). Our pre-installed middleware policy contained 111 types and 18 classes for a fine-granular access control to the major system *Services* and *ContentProviders* (e.g., *ContactsProvider*, *LocationManager*, *PackageManager-Service*, or *SensorManager*). These rules (together with the above stated type and object definitions) constitute our *basic policy.* Table 5.2 provides a breakdown of the policy complexities of SE Android, FlaskDroid's basic policy, and different desktop operating systems with SE Linux support by default. Although SELinux policies cannot be directly compared to our policy, since they target desktop operating systems, the difference in policy complexity between the different Android-based systems (including our FlaskDroid) and the desktop operating systems is noticeable. In particular, since the first evaluation of our FlaskDroid, all professionally maintained policies have evolved, however, the policies for the desktop operating systems have grown in absolute numbers one magnitude more than the SE Android (now AOSP) policy, while at the same time the AOSP policy has reached a very fine-tuned (*strict*) policy configuration for the system components [76]. This underlines that the design of mobile operating systems facilitates a clearer mandatory access control architecture (e.g., separation of duties). This profits an easier policy design (as supported also by the experiences from [212, 126, 128]).

Lesson learned. A particular lesson learned from deriving the basic policy in FlaskDroid is that several permissions operationally depend on each other. For instance, the permission check for starting an *Activity* followed almost always a permission check for *queryIntentActivity* to resolve the target *Activity.* Thus, a launcher app which starts other apps would by default require the permission to perform *queryIntentActivity* for

Policy	# Types	# Attributes	# Classes	# Permissions	# Rules
Android Derivatives					
SE Android (Master branch, checkout 12/04/2012)	232	19	84	249	1,359
SE Android (v5.1.1_r4, checkout 07/06/2015)	368	20	87	269	3,923
AOSP (Master branch, checkout 07/06/2015)	557	23	55	202	5,406
FlaskDroid middleware MAC (basic policy from 12/04/2012)	111	9	18	63	109
Linux Distributions					
SELinux reference policy (v2.20120725, no distribution option)	661	132	81	239	278
SELinux Fedora 17 (targeted, policy.27 from 12/04/2012)	3,900	313	83	248	103,235
SELinux Fedora 22 (targeted, policy.29 from 07/06/2015)	4,638	357	83	254	128,273
SELinux CentOS 6.3 (targeted, policy.24 from 12/05/2012)	3,508	277	81	235	275,791
SELinux CentOS 6.6 (targeted, policy.24, from 07/06/2015)	3,851	291	81	237	610,413
SELinux CentOS 7 (targeted, policy.29 from 07/06/2015)	4,620	357	83	255	127,794
SELinux Debian 6.0.6 (targeted, policy.24 from 12/05/2012)	1,285	190	77	229	49,159
SELinux Debian 7.8 (targeted, policy.26 from 07/06/2015)	1,342	201	83	247	57,580
SELinux Debian Sid (targeted, policy.29 from 07/06/2015)	3,945	289	83	251	117,426

Table 5.2: Overview of policy complexity: Different SELinux policies on desktop OSes vs. SELinux-enabled Android derivatives.

any other app in order to function properly. Similarly, the application *Context* class inquires information about its corresponding app from the system *PackageManagerService* upon initialization. Thus all 3^{rd} party apps require in general the permission to perform the *getPackageInfo* operation. These insights can be used to further optimize the system policy (e.g., establishing sets of fundamental privileges for different application categories like *Launcher App*) and we intent to investigate these relations further in future work.

5.8.1.3 3^{rd} Party Policies

The derived basic policy can act as the basis on top of which additional user, 3^{rd} party, and use-case specific policies can be deployed (cf. Section 5.7). In particular, we are currently working on extending the basic policy with types, classes and allow rules for

Attack vector	**Test cases**
Root exploit	mempodroid Exploit
App executed by root	Synthetic Test App
Over-privileged apps and Information-stealing trojans	Known malware Synthetic Test App WhatsApp v2.8.4313 Facebook v1.9.1
Sensory malware	Synthetic Test App emulating [211, 19, 161]
Confused deputy attack	Synthetic Test App
Collusion attack	Synthetic Test Apps emulating [161]

Table 5.3: List of attacks considered in our testbed.

popular apps, such as WhatsApp or Facebook, which we further evaluated w.r.t. user's privacy protection (cf. Section 5.8.2). A particular challenge is to derive policies which on the one hand protect the user's privacy but on the other hand preserve the intended functionality of the apps. Since the user privacy protection strongly depends on the subjective security objectives of the user, this approach requires further investigation on how the user can be involved in the policy configuration [218].

However, as discussed in Section 5.6.2.2, multiple policies by different stakeholders with potentially conflicting security objectives require a reconciliation strategy. Devising a general strategy applicable to all use-cases and satisfying all stakeholders is very difficult, but use-case specific strategies are feasible [154, 94]. In our implementation, we opted for a consensus approach, which we successfully applied during implementation of our use-cases (cf. Section 5.7). We explained further possible strategies in Section 5.6.3.2. More specifically, we opted for a consensus approach with deny precedence (i.e., the access is denied if one policy denies it) and a mandatory system policy (cf. Section 5.6.2.2) to resolve conflicts between 3^{rd} party policies and the system policy. Thus, 3^{rd} party policies are generally independent of the system policy and they can only refine the access rights of their host app as granted by the system policy. Resolving conflicts between two 3^{rd} party policies is out of scope of our conflict resolution, since we cannot interpret the high-level security objectives that the respective 3^{rd} parties had in mind when developing their policies. In this case we apply again a consensus strategy, i.e., both 3^{rd} party policies must allow an operation, otherwise access is denied.

5.8.2 Effectiveness

We decided to evaluate the effectiveness of FlaskDroid based on empirical testing using the security models presented in our Use-cases Section 5.7 as well as a testbed of known malware retrieved from [3, 23] and synthetic attacks (Table 5.3 provides an overview). Alternative approaches like static analysis [41] would benefit our evaluation but are out of scope of this work and will be addressed separately in future work.

5.8.2.1 Evaluation of Kernel Layer Security

SE Android successfully mitigates the effect of the *mempodroid* root exploit. While the exploit still succeeds in elevating its process to root privileges, the process remains constrained to the limited privileges granted to the root user by the underlying SE Android policy [175, 174, 173].

5.8.2.2 Evaluation of Middleware Layer Security

Over-privileged and information stealing apps. Android's permission framework prior to the upcoming Android version 6 release is too coarse grained and inflexible to allow end-users to fine-tune the access applications have to their private information. For example, the `READ_CONTACTS` permissions grants an app access to the entire contacts database, and there is no possibility to further restrict access to specific data, such as names, email addresses or phone numbers.

Examples: Designated information-stealing malware, such as *Android.LoozFon* [119] and *Android.Enesoluty* [182], use the permissions granted to them by the user to exfiltrate sensitive information (e.g., contacts and device information like phone numbers, IMEI/IMSI number etc.).

Over-privileged apps, in contrast, are not malicious by design, but request more permissions than required for their correct functionality [147]. Older versions of the Facebook app, for example, temporarily requested the `READ_SMS` permission from the user without strictly requiring it. Another example is the WhatsApp messenger, although this is not strictly speaking over-privilege in light of the coarse-grained permission definitions: WhatsApp (similar apps) request access to the entire contacts database by asking for the `READ_CONTACTS` permission, although it only needs access to the contacts' phone numbers and names for correct functionality.

Mitigation: We verified the effectiveness of FlaskDroid against over-privileged apps using (a) a synthetic test app which uses its permissions to access the *ContactsProvider*, the *LocationManager* and the *SensorManager* as 3^{rd} party apps would do; (b) malware such as Android.LoozPhone and Android.Enesoluty which steal user's private information; and (c) unmodified apps from Google Play, including social networking apps like the popular WhatsApp messenger and Facebook apps.

In all cases, a corresponding policy on FlaskDroid successfully and gracefully prevented the apps and malware from accessing privacy critical information from sources such as the *ContactsProvider* or *LocationManager*.

However, while blocking access to these system services did not cause application crashes, the applications often exhibited impeded functionality. For instance, prohibiting that the WhatsApp app can access the *ContactsProvider* and thus upload our contacts' phone numbers to its server, resulted in an empty WhatsApp contacts list and thus prevented us from messaging with our contacts. This effect concurs with the observations made in [93, 103] and motivates a more fine-grained enforcement within *ContentProviders* to allow the user to configure policies to share data (e.g., contacts phone numbers and names) and simultaneously protect her sensitive data (e.g., all other contacts data like email addresses). Listing 5.11 presents an example on how such fine-grained access control in the system *ContactsProvider* can be implemented in our policy language.

Listing 5.11: Policy snippet showing access control for the system *ContactsProvider*.

```
[...]

attribute allContactsData_t;

type contacts_email_t, allContactsData_t;
type contacts_postal_t, allContactsData_t;
type contacts_name_t, allContactsData_t;
type contacts_number_t, allContactsData_t;

allow {app_trusted_t} allContactsData_t: contactsProvider_c {query ↪
    insert delete update};
allow {app_whatsapp_t} {contacts_name_t, contacts_number_t}: ↪
    contactsProvider_c {query};

[...]
```

Sensory malware. This class of malware uses the device's current context in combination with data obtained from onboard sensors, such as the acceleration sensor data, to derive and exfiltrate privacy-sensitive information.

Examples: The *TouchLogger* [19] and *TapLogger* [211] attacks use information from the acceleration sensor to derive which keys the user has pressed on the on-screen keyboard. This information can be used to indirectly retrieve passwords the user has entered using the touchscreen.

Another example is the *Soundcomber* trojan [161], which extracts credit card information from recorded calls. By analyzing the current phone state, SoundComber is aware of incoming and outgoing calls. It records and analyzes the call and is able to extract credit card information from calls to telephone banking services.

Mitigation: We emulate the behaviour of such malware using a synthesized test app. To mitigate the attack, we deployed a context-aware FLASKDROID policy that causes the *SensorManager USOM* to filter acceleration sensor information delivered to registered *SensorListeners* while the on-screen keyboard is active. Listing 5.12 presents the corresponding policy. The on-screen keyboard state is mapped to a boolean `keyboard_b` by a *Context Provider* dedicated to monitoring which application is shown in foreground on the screen.[11] While `keyboard_b` is `true`, access to sensor data exposed by the *SensorManager USOM* is denied to all apps. Similarly, a second policy rule prevents the SoundComber attack by denying any access to the audio record functionality implemented in the *MediaRecorderClient USOM* while a call is in progress The call state is also mapped to a boolean `telephony_b` that is set by a *Context Provider* monitoring the call state in the *TelephonyManager Service*.

[11]This required an extension to the *ActivityManagerService* to provide this information to the *Context Provider*, since by default the application framework does not divulge this information.

Listing 5.12: Policy snippet for context-aware access control on the *SensorManager* and *MediaRecorder*.

```
[...]

bool keyboard_b = false;
bool telephony_b = false;

if(~keyboard_b)
{
    allow app_allApp_t allSensorData_t: sensorManager_c {update};
};

if(~telephony_b)
{
    allow app_allApp_t audio_t: mediaRecorderClient_c {record};
};

[...]
```

Confused deputy and collusion attacks. A confused deputy on Android is an app which holds the necessary permissions to access sensitive data or *Services*, and that is vulnerable to being misused for malicious intent by an unprivileged app (e.g., via an unprotected interface of the deputy app). Collusion attacks are based on two (or more) inconspicuous apps that for themselves are not security-critical (e.g., limited set of permissions), but collaborate to implement malicious behavior (e.g., merge their permission sets).

Examples: Confused deputies have been shown to be a wide-spread problem among Android apps. For example, the Settings Widget of previous Android versions contained a confused deputy in a *Broadcast Receiver* component of the *SettingsAppWidgetProvider* class [148]. By sending a *Broadcast Intent* with a specific action string, a non-privileged app can enable or disable the GPS receiver and Wifi connections. Further work has identified more confused deputies on Android [44, 215, 125].

While Collusion attacks are currently mainly an academic topic, Android's system design has been shown to be be vulnerable to these attacks [161, 30, 117]. The previously mentioned SoundComber trojan [161] uses two apps that exchange data over a covert channel, for instance, the system audio volume settings. The first app has the `READ_PHONE_STATE` and `RECORD_AUDIO` permission. It records and analyzes phone calls and encodes sensitive information into volume settings. The second app possesses the `INTERNET` permission. It monitors volume setting changes, decodes the sensitive data, and sends it to a remote server.

Mitigation: The previously described confused deputy attack in the *Settings-AppWidgetProvider* class is addressed by our fine-grained access control rules on ICC. Listing 5.13 shows a policy that restricts the app types that may send (*Broadcast*) *Intents* reserved for system apps to the type `android_t`. By limiting the allowed set of *Broadcast Intent* senders and receivers, unprivileged apps are prevented from controlling the GPS and Wifi state.

Listing 5.13: Policy snippet for restricting the root user on the middleware layer.

```
[...]

appType android_t
{
  /* All packages under this UID */
  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;
};

intentType systemAppWidgetIntent_t
{
  Action:has_action=
    android.appwidget.action.APPWIDGET_UPDATE |
    android.appwidget.action.APPWIDGET_DISABLED |
    android.appwidget.action.APPWIDGET_ENABLED;
  Categories:has_category=
    android.intent.category.ALTERNATIVE;
};

allow android_t systemAppWidgetIntent_t: broadcast_c {send sendSticky ↷
    receiveSticky registerReceiver unregisterReceiver};

[...]
```

Collusion attacks are in general more challenging to handle, especially when covert channels are used for communication. Similar to the mitigation of confused deputies, a FlaskDroid policy was used to prohibit ICC between colluding apps based on specifically assigned app types. However, to address collusion attacks *efficiently*, more flexible policies are required. We already discussed in Section 5.6.2.4 a possible approach to instantiate the XManDroid security model [P5] based on our *Context Providers* and we elaborate in the subsequent Section 5.8.2.3 on particular challenges for improving the mitigation of collusion attacks.

5.8.2.3 Challenges and Trusted Computing Base

Information flows within apps. Like any other access control system, e.g., SELinux, exceptions for which enforcement falls short concern attacks which are licit within the policy rules. Such shortcomings may lead to unwanted information leakage. A particular challenge for addressing this problem and controlling access and separation (*non-interference* [68]) of security relevant information are information flows within apps. Access control frameworks like FlaskDroid usually operate at the granularity of application inputs/outputs but do not cover the information flow within apps. For Android security, this control can be crucial when considering attacks such as collusion attacks and confused deputy attacks. Specifically for Android, taint tracking based approaches [46, 93, 156] and extensions to Android's IPC mechanism [36] have been proposed. To which extent these approaches could augment the coverage and hence effectiveness of FlaskDroid has to be explored in future work. An alternative approach to taint tracking from the desktop operating system area, called *SIESTA* [92], that is also motivated by the inability of OS-based access control to reason about how applications treat sensitive data, intertwines language-based security with MAC. In SIESTA, SELinux labels are communicated to application processes on data access and it leverages a policy-enhanced version of *JiF* [91] to ensure that labeled data flows within applications in compliance with the MAC policy. Since Android applications are authored in Java (with exception of included native code), this approach might be feasible on Android as well, but requires further investigation on supporting language-based security on Android.

Trusted Computing Base. Since SE Android executes as part of the kernel, it is susceptible to kernel-exploits, including rootkits [14, 32], or to physical attacks that can exchange the deployed system images [197] and hence the trusted computing code base. While FlaskDroid does not make any attempts beyond access control policies to protect the kernel integrity, orthogonal work [6, 63] has introduced solutions to ensure kernel code integrity at runtime. Moreover, our middleware extensions might be compromised by attacks against the process in which they execute. Currently our *SecurityServer* executes within the scope of the rather large Android system server process. Separating the *SecurityServer* as a distinct system process with a smaller attack surface (smaller TCB) can be efficiently accomplished, since there is no strong functional inter-dependency between the system server and *SecurityServer*.

5.8.3 Performance Overhead

Middleware layer. We evaluated the performance overhead of our architecture based on the *No-allow-rule* policy and the basic policy presented in Section 5.8.1 using a Samsung Galaxy Nexus device running FlaskDroid. Table 5.4 presents the mean execution time μ and standard deviation σ for performing a policy check at the middleware layer in both policy configurations (measured in μs) as well as the average memory consumption (measured in *MB*) of the process in which our *User Space Security Server* executes (i.e., the system server). Average execution time and standard deviation are the amortized values for both cached and non-cached policy decisions. The margins of error for the mean execution times are for the 95% confidence interval.

In comparison to permission checks on a vanilla Android 4.0.4 both the imposed runtime and memory overhead are acceptable. The high standard deviation is explained by varying system loads, however, Figure 5.5 presents the cumulative frequency distribution for our policy checks and shows that 99% of the policy checks with our basic policy are performed in less than $1.5ms$.

In comparison to closest related work [P5, P4] (cf. Section 5.9), FlaskDroid achieves a very similar performance. Table 5.5 provides an overview of the average performance overhead of the different solutions. TrustDroid [P4] profits from the very static policies it enforces, while FlaskDroid slightly outperforms XManDroid [P5].

	μ (in μs)	σ (in μs)	memory (in MB)
FlaskDroid v4.0.4			
No-allow-rule	329.505±33.9	780.563	15.673
Basic policy	452.916±208.58	4,887.24	16.184
Vanilla Android v4.0.4			
Permission check	330.800±185.64	8,291.805	15.985

Table 5.4: Runtime and memory overhead on Android v4.0.4 (on Samsung Galaxy Nexus). Margins of error are for the 95% confidence interval.

	μ (in ms)	σ (in ms)
FlaskDroid (Basic policy)	0.452	4.887
XManDroid [P5] (Amortized)	0.532	2.150
TrustDroid [P4]	0.170	1.910

Table 5.5: Performance comparison to related works.

However, it is hard to provide a completely fair comparison, since both TrustDroid and XManDroid are based on Android 2.2 and thus have a different baseline measurement. Both [P5, P4] report a baseline of approximately $0.18ms$ for the default permission check, which differs from the $0.33ms$ we observed in Android 4.0.4 (cf. Table 5.4).

Re-evaluation of middleware layer. Post-publication we ported FlaskDroid from Android v4.0.4 to Android v5.1.1 and re-evaluated the middleware performance overhead on an LGE Nexus 5 device. Table 5.6 presents the new mean execution time μ and standard deviation σ for performing a policy check at the middleware layer in both policy configurations. Average execution time and standard deviation are again the amortized values for both cached and non-cached policy decisions. The margins of error for the mean execution times are also again for the 95% confidence interval.

Although the porting required the definition of 79 new allow rules, the *mean* performance overhead between the two versions stayed almost identical. However, as Figure 5.6 shows, the cumulative frequency distribution of FlaskDroid's performance overhead on Android v5.1.1 has a less favorable shape, in particular the threshold for the 99th percentile has shifted to $6.3ms$. This motivates the need for future performance oriented engineering of our *USSS*. Performance has not been a focus of the current prototypical implementation and hence left room for performance enhancements (for instance, better performing cache algorithms for an access vector cache).

Moreover, Table 5.6 gives in comparison to Table 5.4 the impression that the performance of the Android permission check has significantly improved between versions and test devices. However, in fact the curves for the permission check in Figures 5.5 and 5.6 have nearly identical shapes. The difference in the mean execution time for the permission check is explained by an extreme performance jitter for the permission check on Android v4.0.4 (reflected also in the high standard deviation σ in Table 5.4 and the slower convergence of the corresponding curve in Figure 5.5 to the 100% boundary). In our re-evaluation, we could re-affirm this jitter on vanilla Android v4.0.4 on a Samsung Galaxy Nexus test device.

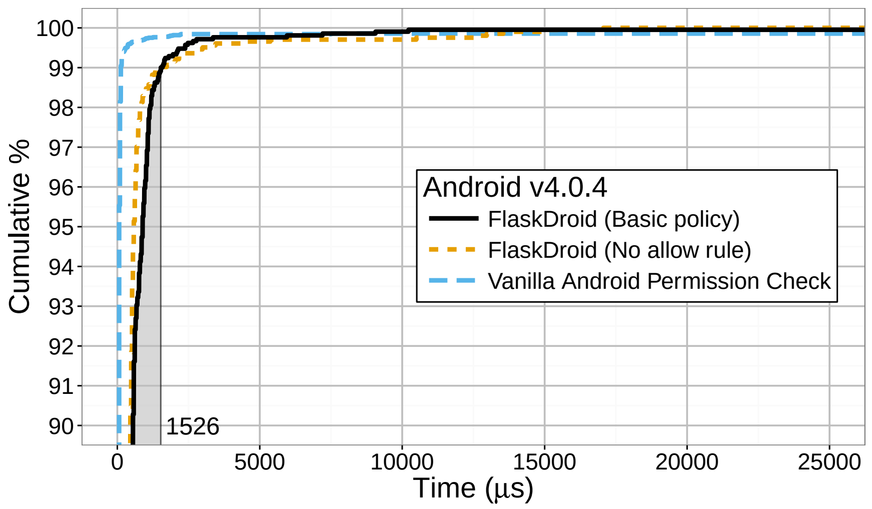

Figure 5.5: Relative cumulative frequency distribution of the performance overhead with a basic policy (solid line) and no allow rules (short-dashed lined) of FlaskDroid on Android v4.0.4 (Samsung Galaxy Nexus device). Performance overhead of permission checks on vanilla Android v4.0.4 is provided for comparison (long-dashed line). The grey shaded area represents the 99th percentile for the basic policy.

	μ (in μs)	σ (in μs)
FlaskDroid v5.1.1		
No-allow-rule	381.58±33.29	2249.74
Basic policy	426.71±54.61	3780.35
Vanilla Android v5.1.1		
Permission check	44.95±21.78	861.53

Table 5.6: Runtime overhead on Android v5.1.1 (on LGE Nexus 5). Margins of error are for the 95% confidence interval.

Kernel layer. The impact of SE Android on Android system performance has been evaluated previously by its developers [175, 173]. Since we only minimally modify the default SE Android policy to cater for our use-cases (e.g., new booleans), the negligible performance overhead reported in [175, 173] still applies to our implementation.

5.9 Related Work

In this section we provide an overview of related work for closes related work on mandatory access control and Android security extensions. We first discuss in Section 5.9.1 the related work at the time this work was conducted in order to underline the novelty of our results. We then discuss in Section 5.9.2 further related work that has been published after our work had been presented.

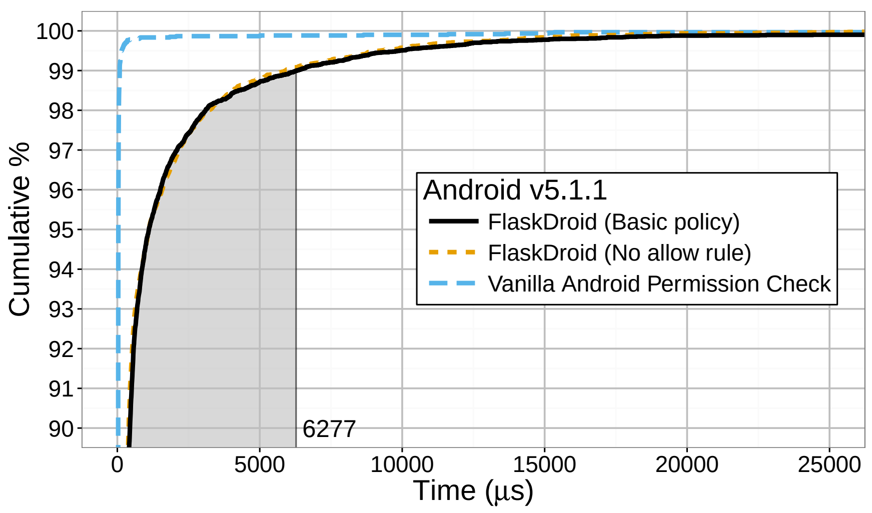

Figure 5.6: Relative cumulative frequency distribution of the performance overhead with a basic policy (solid line) and no allow rules (short-dashed lined) of FLASKDROID on Android v5.1.1 (LGE Nexus 5 device). Performance overhead of permission checks on vanilla Android v5.1.1 is provided for comparison (long-dashed line). The grey shaded area represents the 99th percentile for the basic policy.

5.9.1 Status Quo at Time of Publication

5.9.1.1 Mandatory Access Control

Flux Advanced Security Kernel (Flask) [179] is an architectural framework for Mandatory Access Control (MAC). It proposes a security architecture that decouples policy enforcement from the security policy itself thus providing for a generic architecture where multiple, dynamic security policies can be supported.

The most prominent MAC solution is SELinux [114] and we elaborated on it in detail in Section 5.4.1. Specifically for mobile platforms, related work [166, 175, 212, 126, 128] has investigated the placement of SELinux enforcement hooks in the operating system and user-space services on Android [166, 175] (cf. Section 2.1), OpenMoko [126] and the LiMo (Linux Mobile) platform [212]. Our approach follows along these ideas but for the Android middleware.

Also TOMOYO Linux [87], a path-based MAC framework, has been leveraged in Android security extensions [P5, P4]. Although TOMOYO supports more easily policy updates at runtime and does not require extended file system attributes, SELinux is more sophisticated, supports richer policies, and covers more object classes [184].

However, as we state in Section 5.5, low-level MAC alone is insufficient. In this work we show how to extend the SE Android security architecture into the Android middleware layer for policy enforcement.

Mandatory Access Control enforcement has also been shown to be beneficial in measuring and attesting the mobile platform's integrity state. The *PRIMA* [99] concept reduces measuring the platform state from cryptographically hashing the binary code and libraries to cryptographically measuring the deployed access control policy. An attestor can deduce from this policy whether higher-integrity applications can be illegally accessed by lower-integrity applications. PRIMA has been successfully applied to a Linux-based mobile platform [128] using SELinux and the Clark-Wilson-lite integrity model [167, 22]. In particularly in the mobile context, integrity attestation is valuable, for instance, when controlling devices' remote access to networks or data [157]. Using FlaskDroid, the PRIMA concept could be applied to attesting possible inter-application data flows and, thus, allow reasoning about the device state in a fashion similar to [128].

Lastly, recent research [185] has demonstrated the ability of mobile botnets to degrade the service of cellular network services. This underlines the importance of mobile end-device security in not only protecting the end-user security but also the networks the devices are connected to. FlaskDroid provides a suitable foundation in enforcing corresponding security policies on the end-devices.

5.9.1.2 SE Android MMAC

The SE Android project was recently extended by different mechanisms for mandatory access control at Android's middleware layer [172], denoted as *MMAC*:

Permission revocation. This is a simple mechanism to dynamically revoke permissions by augmenting the default Android permission check with a policy driven check. When necessary, this additional check overrules and negates the result of the default check.

However, this permission revocation is in almost all cases unexpected for app developers, which rely on the fact that if their app has been installed, it has been granted all requested permissions. Thus, developers very often omit error handling code for permission denials and hence unexpectedly revoking permissions easily leads to application crashes.

In FlaskDroid, policy enforcement also effectively revokes permissions. However, we use *USOMs* which integrate the policy enforcement into the components which manage the security and privacy sensitive data. Thus, our *USOMs* apply enforcement mechanisms that are *graceful*, i.e., they do not cause unexpected behavior that can cause application crashes. Related work (cf. Section 5.9.1.3) introduced some of these graceful enforcement mechanisms, e.g., filtering table rows and columns from *ContentProvider* responses [218, 24, 93, T2, P5, P4].

Intent MAC. Intent MAC protects with a white-listing enforcement the delivery of *Intents* to *Activities*, *Broadcast Receivers*, and *Services*. Technically, this approach is similar to prior work like [218, P5, P4]. The white-listing is based on attributes of the *Intent* objects (e.g., the value of the action string) and the security type assigned to the *Intent* sender and receiver apps.

In FlaskDroid, we apply a very similar mechanism by assigning *Intent* objects a security type, which we use for type enforcement on *Intents*. While we acknowledge, that access control on *Intents* is important for the overall coverage of the access control, Intent MAC alone is insufficient for policy enforcement on inter-app communications. A complete system has to consider also other middleware communications channels, such as Remote Procedure Calls (RPC) to *Service* components and to *ContentProviders*.

By instrumenting these components as *USOMs* and by extending the AIDL compiler (cf. Section 5.6.2) to insert policy enforcement points, we address these channels in FlaskDroid and provide a non-trivial complementary access control to Intent MAC.

Install-time MAC. Install-time MAC performs, similar to *Kirin* [44], an install-time check of new apps and denies installation when an app requests a defined combination of permissions. The adverse permission combinations are defined in the SE Android policy.

While FlaskDroid does not provide an install-time MAC, we consider this mechanism orthogonal to the access control that FlaskDroid already provides and further argue that it could be easily integrated into existing mechanisms of FlaskDroid (e.g., by extending the install-time labeling of new apps with a blacklist-based rejection of prohibited app types).

5.9.1.3 Android Security Extensions

In the recent years, a number of security extensions to Android have been proposed.

Different approaches [139, 133, 24, 140] add mandatory access control mechanisms to Android, tailored for specific problem sets such as providing a DRM mechanism (*Porscha* [139]), providing the user with the means to selectively choose the permissions and runtime constraints each app has (*APEX* [133] and *CRePE* [24]), or fine-grained, context-aware access control to enable developers to install policies to protect the interfaces of their apps (*Saint* [140]). Essentially all these solutions extend Android with MAC at the middleware layer. The explicit design goal of our architecture was to provide an ecosystem that is flexible enough to instantiate those related works based on policies (as demonstrated in Section 5.7 at the example of *Saint*) and additionally providing the benefit of a consolidated kernel-level MAC.

The authors of *Porscha* [139] propose a DRM mechanism to enforce access control on specifically tagged data, such as SMS or e-mails. However, this approach is limited to isolate data assets, but is not suitable as a generic access control framework, e.g., considering interfaces as objects. By implementing the corresponding USOMs for the required data providers (e.g. SMS) and services, FlaskDroid allows us to implement a solution comparable to Porscha based on subject/object types instead of tagged data.

The pioneering framework *TaintDroid* [46] introduced the tracking of tainted data from sensible sources on Android and successfully detected unauthorized information leakage. The subsequent *AppFence* architecture [93] extended TaintDroid with checks that not only detect but also prevent such unauthorized leakage. However, both TaintDroid and AppFence do not provide a generic access control framework. Nevertheless, future work could investigate their applicability in our architecture, e.g., propagating the security context of data objects. The general feasibility of such "context propagation" has been shown in the *MOSES* [156] architecture. In MOSES, apps and data are compartmentalized into different *security profiles* (e.g., `work` and `private`) and MOSES enforces isolation of these profiles. To this end, it applies labeling of data objects with their assigned profile and introduced policy enforcement points in Android middleware services and libraries (e.g., `LibBinder`, `Socket` class, or `OSFileSystem`) for fine-grained access control based on the labels. MOSES relies on the TaintDroid framework to propagate the labels across process boundaries and thus addresses the problems discussed in Section 5.8.2.3.

To achieve policy enforcement for 3^{rd} party apps *without* the need to modify the Android operating system, some recent works leverage so-called *Inlined Reference Monitors* (IRM) [210, 7, 101, 33, 34, 195]. IRM places the policy enforcement code directly in the 3^{rd} party app instead of relying on a system centric solution. An unsolved problem of *inlined* monitoring in contrast to a system-centric solution is that the reference monitor and the potentially malicious code share the same sandbox and that the monitor is *not* more privileged than the malicious code[86]. This means that native code, which is by design supported in Android, can be maliciously used to access the IRM memory region and disable it at runtime.

The closest related work to FlaskDroid with respect to a two layer access control are our previous XManDroid [P5] and TrustDroid [P4] architectures. Both leverage TOMOYO Linux as kernel-level MAC to establish a separate security domain for business apps [P4], or to mitigate collusion attacks via kernel-level resources [P5]. Although they cover MAC enforcement at both middleware and kernel level, both systems support only a very static policy tailored to their specific purposes and do not support the instantiation of different use-cases. In contrast, FlaskDroid can instantiate the XManDroid and TrustDroid security models by adjusting policies. For instance, different security types for business and private apps could be assigned at installation time, and boolean flags can be used to dynamically prevent two apps from communicating if this would form a collusion attack.

5.9.2 Related Work Post-Publication

Since this work has been published, Google has integrated SELinux support into the Android code base [72, 73, 74, 75, 76]. Main contributor to this work was the SE Android project [175, 132], which also formed the foundation for our kernel-level MAC in FlaskDroid. With Android version 5, the targeted policy for Android has evolved to the point that all (system) services and daemons are executed in their own security domain and that the unconfined domain is removed. SELinux has also been used in the past in commercial products, such as *Samsung Knox* [159], to isolate different security domains (e.g., personal and business).

A very recent work, *EASEAndroid* [194], tackles the problem of (semi-)automatically analyzing the audit logs of SELinux access control enforcement on real devices at large-scale and thus helping security analysts in better understanding and refining the deployed policies. The main challenge was to analyze a huge number of logs that contain a mixture of benign and malicious behavior and to correctly distinguish those behaviors and label unknown subjects and behaviors. To this end, the authors devised a tool based on a semi-supervised classifier and policy refiner that mimics the methodology of human security analysts.

Like other access control solutions on Android, FlaskDroid's enforcement can negatively affect applications through denied inter-component communication. Recent work [103] has quantified the effects of denying applications access to application framework APIs and as lesson learned from our TrustDroid and XManDroid frameworks as well as related work [218, 93], the enforcement hooks in FlaskDroid hardcode logic to make denials more graceful for applications. This prevents apps from crashing unexpectedly, however, still renders them dysfunctional (e.g., when mission-critical data is denied).

Other recent work investigated the implications of finer-grained access controls for app developers and the end-users [59], with focus on the *Internet* permission. Thus, this work relates to FlaskDroid's feature of allowing app developers to opt-in to FlaskDroid by attaching custom policies to their apps and instrumenting their apps as *USOMs*. Through an empirical study of the Internet communication behavior of more than 1,000 apps using a symbolic executor, the authors come to the conclusion that fine-access control is feasible and practical, however, it is paramount that both users and developers are better supported in selecting and defining fine-grained permissions. A conclusion that is also backed up by previous study results on permission usage in apps [147].

As for TrustDroid and XManDroid, an inherent limitation of the Android application model (which FlaskDroid did not aim at changing) is enforcement at the granularity of application sandboxes (i.e., UIDs). Similarly, new approaches such as per-component access control in *Compac* [196] or per-ad-lib access control as in *AdSplit* [170] and *AdDroid* [144] could improve effectiveness of the enforcement of FlaskDroid's middleware extensions.

Although FlaskDroid supports context-aware policies, an open challenge is efficiently defining security- and privacy-relevant contexts, which are very often highly subjective to the end-user. *ConXsense* [122] addresses this problem by utilizing machine learning to automatically classify security- and privacy-relevant contexts based on their properties. The authors also present the deployment of their context-sensing framework on top of our FlaskDroid solution in order to implement two use-cases that protect against sensory malware and against device misuse through context-aware device lock.

FlaskDroid's design has also been improved by our Android Security Framework [P2] (cf. Chapter 6) and by the independently, concurrently developed *Android Security Modules* [90] framework. Both provide a programmable security API to instantiate various security models programmatically and thus avoid limiting the policy author to one specific policy language as FlaskDroid does. Moreover, in our Android Security Framework (ASF) work, we demonstrate how FlaskDroid can be effectively instantiated as a use-case on top of ASF.

Furthermore, our recent *Boxify* [S1] app virtualization framework offer a new deployment opportunity for FlaskDroid. Currently, FlaskDroid is implemented as a modified Android software stack. However, solutions that are based on modified software stacks and require a custom ROM to be deployed are notorious for low-acceptance for real-life deployment due to the high technical expertise such deployment requires. Based on Boxify, at least FlaskDroid's middleware extensions could be deployed efficiently as part of Boxify's virtualization layer, while FlaskDroid's kernel-level MAC requires more extensive extensions to Boxify's syscall interceptor to virtualize type enforcement. On the plus side, a Boxify-based implementation of FlaskDroid could also provide additional benefits, such as a per-component access control as in Compac [196].

Lastly, new attack vectors against Android's system integrity have been discovered. In a *Pileup* attack [209], an attacker can escalate the privileges of his app during a system upgrade. The attacker app declares in its application manifest properties (such as permissions or shared UIDs) that are only valid on a higher version of the Android OS. When the system is upgraded, errors in the update logic of *PackageManagerService* grant those illegally requested properties to the attacker app. While FlaskDroid's type assignment to apps is not necessarily affected by the illegal permissions an app gains through Pileup, the shared UID grabbing and data contamination effects of Pileup can undermine FlaskDroid's access control enforcement. Another recently discovered attack vector are unprotected device driver interfaces induced by vendor customizations on customized Android ROMs [214]. As in Android and SE Android, a base policy (like the one we derived for our evaluation) does not cover any customized interfaces and it is the responsibility of the vendor to adapt the base policy to the customized interfaces. Failure in adapting the policy leads to gaps in the access control enforcement and, as shown in this study [214], to potentially severe compromise of the platform integrity or user's privacy (e.g., unauthorized access to the device camera or monitoring user input to the touch screen, which both can lead to follow-up attacks like *PlaceRaider* [183] or keylogging [19, 211]).

5.10 Conclusion

In this work, we presented the design and implementation of FlaskDroid, a policy-driven generic two-layer MAC framework on Android-based platforms. We introduced our efficient policy language that is tailored for Android's middleware semantics. We show the flexibility of our architecture through policy-driven instantiations of selected security models, including related work and privacy-enhanced system components. We demonstrated the applicability of our design by prototyping it on Android version 4.0.4. Our evaluation shows that the clear API-oriented design of Android benefits the effective and efficient implementation of a generic mandatory access control framework like FlaskDroid.

6

Android Security Framework

Extensible Multi-Layered Access Control on Android

6.1 Motivation

For several decades now, the need for operating system security mechanisms to provide strong security and privacy guarantees has been well understood [110, 158, 115, 10]. Yet, recent classes of attacks against smartphone end-user's privacy and security [84, 217, 148, 21] have shown that the fairly new smart device operating systems fail to provide these strong guarantees, for instance, with respect to access control or information flow control. To remedy this situation, security research has proposed a wide spectrum of security models and extensions for mobile operating systems, most of them for the popular open-source Android OS. These extensions include context-related access control [24], developer-centric security policies [140], and dynamic, fine-grained permissions [218, 101, 7]. They also comprise security models [P4, 156, 175, P3] such as domain isolation and type enforcement, which are usually at the heart of enterprise and governmental security solutions.

6.2 Problem Description

However, the lack of a comprehensive security API for the development and modularization of security extensions on Android has created the unsatisfactory situation that all of these novel and warranted security models are either provided as model-specific patches to the Android software stack, or they became an integrated component of the Android OS design [175]. When considering the body of literature on established security frameworks, such as *Linux Security Modules* (LSM) [206] or the *BSD MAC Framework* [198], their history has taught that the need to patch the OS or the hardwiring of a specific security model impairs both the practical and theoretical benefits of security solutions. First, there is in general no consensus on the *"right"* security model, as demonstrated by the broad range of Android security extensions [24, 140, 7, 218, P4, 175]. Thus, OS security mechanisms should not limit policy authors to one specific security model by embedding it into the OS design. Second, providing security solutions as *"security-model-specific Android forks"* impedes their maintainability across different OS versions, because every update to the Android software stack has to be re-evaluated for and applied to each fork separately.

6.3 Contributions

In this work, we proposed the design and implementation of ANDROID SECURITY FRAMEWORK (ASF), which allows security experts to develop and deploy their security models in form of modules as part of Android's platform security. This provides the means to easily extend the Android security mechanisms and avoids that security designers have to choose "the right Android security fork" or that the OS vendor has to impose a specific security model. In the design of ASF we transfer the lessons learned and guiding principles from the literature on established OS security infrastructures to Android and intertwine them with new requirements for efficient security policies for multi-tiered software stacks of smart devices. In contrast to concurrent, independent work [90], which introduced extensibility for security *apps* (i.e., add-ons), our design establishes a generic and extensible security framework that allows instantiating security models *by design* as part of Android's platform security and enables not only extending but also replacing Android's default security mechanisms. This is particularly beneficial when tailoring Android for higher-security deployments like enterprise phones, where the default mechanisms are insufficient or even obsolete (e.g., when the IT department is an additional stakeholder that decides on apps' privileges and installation). We make the following contributions:

Policy-agnostic, multi-tiered security infrastructure: The security infrastructure must avoid committing to one particular security model and enable authors of security extensions to develop as well as deploy their solutions in form of code. This requires special consideration of Android's multi-tiered software stack and the dominant programming languages at each layer. For ASF we solve this by integrating security-model-agnostic enforcement hooks into the Android kernel, middleware and application layer and exposing these hooks through a novel security API to module authors.

Enabling mandatory results automata policies: Various Android security solutions realize mandatory results automata policies that not only truncate but also modify control flows through manipulation of the return values. In ASF, the application layer and middleware hooks are specifically designed to allow module authors to leverage the rich semantics of Android's application framework and to implement their security policies as such automata. This required a re-thinking of the "classical" object manager design from the literature by shifting the mandatory results automata logic from the infrastructure into the security modules.

Instantiation of existing security models: We demonstrate the efficiency and effectiveness of our ASF by instantiating different security models from related work on type enforcement [P3, 175] and inlined access control [7] as well as from Android's default security architecture as modules.

Maintenance benefits for security extensions: Our ported security modules show how ASF simplifies maintainability of security extensions across different OS versions by shifting the bulk of effort to the security framework maintainer. This is similar to the maintenance of the application framework for regular apps. Hence, a comparable benefit to regular apps in adaption and stability across OS versions can be expected of security modules.

Research and development benefits: We postulate that developing security solutions against a well documented security API also greatly contributes to *a)* a better understanding and analysis of new security models that form a self-contained unit instead of being integrated to various components of the Android software stack, *b)* a better reproducibility and dissemination of new solutions since modules can be easily shared and instantiated, and *c)* a more convient application of security knowledge to the Android software stack without the requirement to be familiar with the deep technical internals of Android.

6.4 Related Work

We first provide a synopsis of the development of extensible kernel security frameworks and discuss afterwards the current status of security extensions and frameworks for Android.

6.4.1 Extensible Kernel Access Control

The importance of the operating system in providing system security has been very well studied in the last decades [158, 110, 10, 115] and different approaches to extending operating systems with access control and security policies have been explored. These include system-call interposition [58, 150], software wrappers [56], and extensible access control frameworks like *DTE* [8], *GFAC* [1], and *Flask* [179]. To realize these solutions, DTE has been provided as a patch to the UNIX system [9], while GFAC and Flask have been implemented as patches to the Linux kernel by the RSBAC [141] and SELinux [114] projects.

However, this led to an intricate situation: On the one hand, maintaining these solutions as patches incurred high maintenance costs for adapting the patches to kernel changes. On the other hand, none of these solutions was included in the vanilla kernel because this would constrain security policy authors to one specific security model. This constrain would be unsatisfying since there exists in general no consensus on the "right" security model. To remedy this situation, extensible security frameworks have been proposed [206, 198] that allow the extension of the system with trusted code modules that implement specific security models. Module authors are supported with an API that exposes kernel abstractions as well as operations and facilitates the implementation of the desired security architecture and model. The results of this research have been integrated into the mainline kernels as the *Linux Security Modules* framework (LSM) [206] and the *BSD MAC Framework* [198]. Additionally, different access control models, such as SELinux type enforcement [176] or TOMOYO path-based access control [87], have been ported as modules on top of the LSM and the BSD MAC framework.

6.4.2 Android Security

Closest to our approach is the independently and concurrently developed *ASM* [90], which also provides a programmable interface for security extensions. In contrast to ASF, however, it targets "security apps" added in addition to the default Android security architecture. As a consequence, ASM has to address the intricate problem of including untrusted code into highly-privileged context for access control enforcement and consolidating it with existing policies. It avoids this Gordian knot through a trade-off between policy expressiveness and sandboxing of security apps. In contrast, our ASF framework resides beneath the default Android security framework and hence allows instantiation of security models that complement or even substitute parts of the default platform security (see Section 6.7). Hence, ASM can even be implemented as a module in ASF. By definition, we must trust the developer of security solutions for ASF.

In recent years, Android's security has been quite scrutinized, and a wide spectrum of security extensions has been brought forward. To name a few: *CRePE* [24] provides a context-related access control, where the context can be, e.g., the device's location. *Saint* [140] enables developer-centric policies that allow app developers to ship their apps with rules that regulate the app's interactions with other apps. Different approaches to more dynamic and fine-grained permissions have been proposed based on system-centric enforcement (e.g., *TISSA* [218]) or inlined reference monitors (*Dr. Android and Mr. Hide* [101] or *AppGuard* [7]). *XManDroid* [P5] enforces Chinese Wall policies to prevent confused deputy and collusion attacks. *TrustDroid* [P4] and *MOSES* [156] isolate different domains such as "Work" and "Private" from each other. *SE Android* [175] and *FlaskDroid* [P3] bring type enforcement to Android, where SE Android focuses on the kernel layer and has been partially included into the mainline Android source code, and FlaskDroid extends type enforcement to Android's middleware layer on top of SE Android.

6.5 Requirements Analysis

The current development of Android security extensions has strong parallels to the initial development of the above mentioned Linux and BSD security extensions, since current Android security extensions are provided as patches to the software stack or, in the case of SE Android [175], are embedded into the Android source tree. For the same, above mentioned reasons as for the early Linux and BSD security extensions, this impedes the applicability and adaption of Android security extensions and additionally precludes many of the benefits that a modular composition could offer in terms of maintenance: Embedding SE Android's security model into Android's source tree limits policy authors to the expressiveness and boundaries of type enforcement, whereas provisioning security models and architectures as patches to Android's software stack forces policy authors to chose a solution-specific Android fork. This requires for every version update to the Android OS a re-evaluation and port of each separate fork. Moreover, security solutions cannot be easily compared with each other, because their infrastructures are deeply embedded into the Android software stack.

In this work, we develop in the spirit of the two de facto most established security frameworks, *Linux Security Modules* (LSM) [206] and the *BSD MAC Framework* [198], a generic and extensible ANDROID SECURITY FRAMEWORK that allows the instantiation and deployment of different security models as modules at Android's application layer, middleware, and kernel. The two most important guiding principles from LSM and the BSD MAC framework that govern the design of our ANDROID SECURITY FRAMEWORK are: *(1)* provisioning of policies as code instead of data; and *(2)* providing a policy-agnostic OS security infrastructure. In the remainder of this section, we analyze the requirements and challenges for their transfer to the Android software stack.

6.5.1 Policy as code and not data

The first guiding principle is that policies should be supported as code instead of data (such as rules written in one predetermined policy language). Providing an extensible security framework that supports integration of policy logic as code avoids committing to one particular security model or architecture. For Android, this removes the need to chose a particular extension-specific Android fork or to be limited to one specific security model in the mainline Android software stack. Additionally, developing modules against an OS security API provides the benefits of modularization for developing and maintaining security extensions. This includes, foremost, a higher functional cohesion of security modules and lower coupling with the Android software stack and, hence, can significantly reduce the maintenance overhead of modules, especially in case of OS changes. Moreover, it allows a better dissemination, comparison, and analysis of self-contained security modules.

Transferring this principle to an extensible Android security framework poses the additional requirement to consider the semantics and dominant programming languages of the different layers of Android's software stack. LSM and the BSD MAC Framework, as part of the kernel, support modules written in C and operate on kernel data structures (e.g., file system inodes). While this also applies to the Android Linux kernel, an Android security framework should additionally support modules written for Android's semantically-rich middleware and application layers. That means modules written in Java and operating on application framework classes (e.g., Intents or app components).

6.5.2 Policy-agnostic security infrastructure

The second principle is that the security framework and its API should be policy-agnostic. This means that policy-specific intrusions into the software stack are avoided and policy-specific data structures and logic are confined to security modules. The current Android security extensions, however, explicitly insert enforcement hooks and extensions into Android's system services/apps that are specific to their policy language.

A particular additional requirement for a security framework on Android are enforcement hooks in the middleware and application layer that support *mandatory result automata* (MRA)[39] policies, a less powerful version of *edit automata* [109] policies, as promoted by different solutions [218, P4, 101, 7]. Mandatory results automata, in contrast to truncation automata, can not only abort control flows but also divert or manipulate them through manipulation of return values and, thus, give policy authors a higher degree of freedom in implementing their enforcement strategies. For instance, when querying a *ContentProvider* component, the policy could simply deny access by throwing a Java Exception (truncation), but also modify the return value to return filtered, empty, or fake data (MRA). To technically enable security modules to implement MRA, our design requires a re-thinking of the "classical" object manager vs. policy server design that is used, e.g., in LSM or our FlaskDroid [P3]. Object managers (i.e., enforcement points) are responsible for assigning security labels to the objects that they manage and for both requesting and enforcing access control decisions from the policy server (i.e., policy decision point). Because this design embeds the enforcement logic into the system independently from the security model, it is unfit for realizing mandatory results automata. Thus, our design requires hooks that generically support different enforcement strategies and shift the enforcement and object labelling logic from the object managers to the security modules.

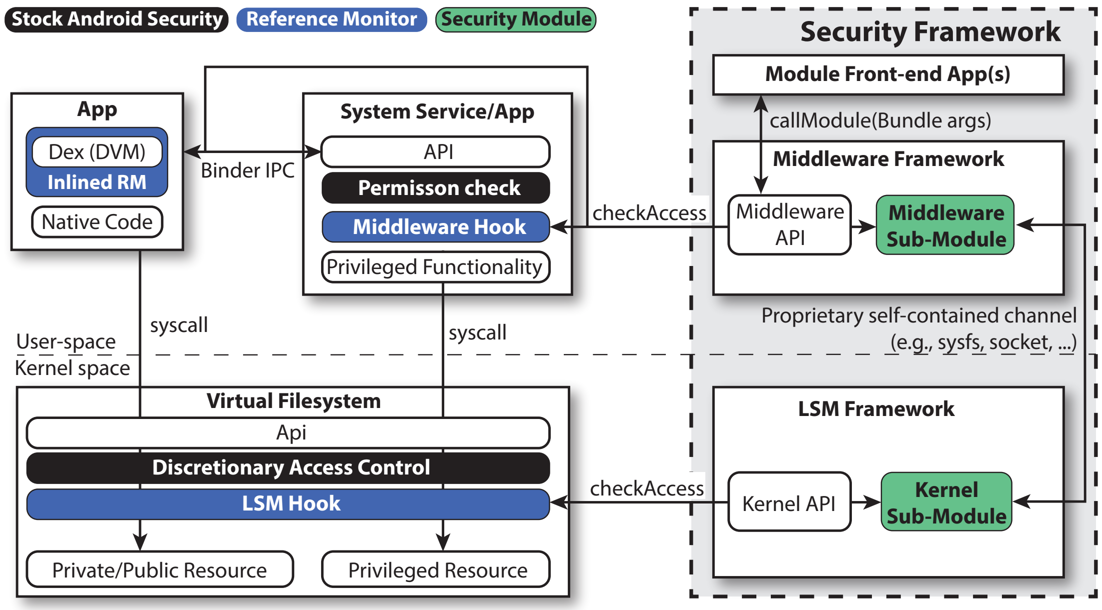

Figure 6.1: ANDROID SECURITY FRAMEWORK architecture.

6.6 ASF Architecture

In the following we present the ASF architecture.

6.6.1 Framework Overview

The basic idea behind our ANDROID SECURITY FRAMEWORK is to extend Android with a new security API that incorporates the design principles explained in Section 6.5. This API allows to easily author, integrate, and enforce generic security policies. This idea continues and extends the design of the *Linux Security Module* framework for the Linux kernel [206] or the *MAC framework* of BSD system kernels [198], but additionally addresses the requirements for the software stacks of smart device operating systems as discussed earlier (cf. Section 6.5). Figure 6.1 provides an overview of our ASF and we explain its building blocks in the following.

6.6.1.1 Reference Monitors

In our design we differentiate between policy enforcing and policy decision making code. For enforcement we use reference monitors [106] at all layers of the Android software stack, i.e., at the application layer, the middleware layer, and the kernel layer. Each reference monitor protects one specific privileged resource and is placed such, that it mediates all access to the resource through the Android API. The benefit of this multi-tiered enforcement is that each reference monitor can operate with the semantics of its respective layer.

6.6.1.2 Security Modules

Security extensions are deployed in the form of code modules and loaded during boot into the security frameworks at the middleware and kernel level. Modules should be signed to ensure their integrity and trustworthiness, and the verification key is embedded in the kernel (or a secure location like a secure execution environment). Each module implements a policy engine that manages its own security policies and acts as policy decision making point. Security modules are integrated into the security frameworks through a security API that exposes objects and operations of the different software stack layers. As part of this API, each module implements an interface for enforcement functions (i.e., functions that make a policy decision for a particular reference monitor in the system), management functions (e.g., module life-cycle events), and other interfaces that will be explained in Section 6.6.2.

To provide a clear separation between policy decision logic using kernel level semantics and logic using middleware/application layer semantics, each module consists of two sub-modules: a KERNEL SUB-MODULE leveraging the already existing Linux Security Module (LSM) infrastructure of the Linux kernel and a MIDDLEWARE SUB-MODULE, for which we designed and implemented a novel security infrastructure at the application and middleware layers.

6.6.1.3 Front-end Apps

To enable user configurable policies or graphical event notifications, modules might want to include user interfaces. To this end, the module developers (or external parties being aware of the modules) can deploy standard Android apps that act as front-end and that communicate through the framework API with the module. We enable such proprietary module interfaces through a *Bundle* based communication protocol. A Bundle is a key-value store that supports heterogenous value types (e.g., Integer and String) and that can be transmitted via Binder IPC between the app process and the module. It is the responsibility of the module to verify that the caller is sufficiently privileged.

6.6.2 Framework Infrastructure

We present now in a bottom-up approach details about the ASF infrastructure that has been prototypcially implemented for Android v4.3 and currently comprises 4,606 lines of code.

6.6.2.1 Kernel Space

At kernel level we employ the existing Linux Security Module (LSM) [206] framework of the Linux kernel. LSM implements an infrastructure for mandatory access control and provides a number of enforcement hooks within kernel components such as the process management or the virtual filesystem. The KERNEL SUB-MODULE is implemented as a standard Linux Security Module that registers through the LSM API for the LSM hooks in the system and that operates with kernel level semantics. KERNEL SUB-MODULE can be an existing Linux security module like SELinux [176], TOMOYO [87], or proprietary

Listing 6.1: Exemplary enforcement functions.

```java
1 public boolean deliverToRegisteredReceiver (Intent intent, ComponentName targetComp, String requiredPermission, int targetUid, int targetPid, String callerPackage, ApplicationInfo callerApp, int callingUid, int callingPid);
2 public Location getLastLocation (Location currentLocation, LocationRequest request, int callingUid, int calingPid);
```

ones [131, 90]. Kernel-level policies form truncation automata that terminate illegal control flows, e.g., on access to files.

Since there might be operational inter-dependencies between the Kernel Sub-Module and user-space processes like the Middleware Sub-Module (e.g., propagation of access control decisions), the kernel module can implement proprietary channels for communication between kernel- and user-space (e.g., sysfs entries).

6.6.2.2 Middleware Layer

At the middleware layer we extended the system services and apps that implement the Android API with hooks that enforce access control decisions made by the Middleware Sub-Module. The middleware security framework is executed as a new Android system service and mediates between our hooks and the Middleware Sub-Module. The hooks are policy-agnostic and not tailored to one specific security model. Each hook takes as arguments all relevant, ambient information of the current control flow that led to the hook's invocation. For instance, Listing 6.1 presents two exemplary hooks in our system: one for the Intent broadcasting subsystem of the *ActivityManagerService* (line 1) and one for the *LocationManager* that implements the location API of Android (line 2). Both provide to the Middleware Sub-Module information about the current caller to the Android API, i.e., App in Figure 6.1 (parameters `callingUid` and `callingPid`). However, all other parameters are specific to the hooks' contexts, e.g., the hook in line 1 provides information about the Intent being broadcast and the app component that should receive this Intent (parameters `targetComp` through `targetPid`). Thus, the hooks support policies that use the rich middleware-specific semantics.

In general, all hooks support truncation automata as policies by either allowing the module to throw exceptions that terminate the control flow and that are returned to the caller of the Android API, or by explicitly requiring a boolean return value that indicates whether the hook truncates the control flow or not (line 1 in Listing 6.1). A subset of the hooks additionally supports mandatory result automata policies, that is the module can modify or replace return values of the Android API function or modify/replace arguments that divert or affect the further control flow after the hook. For instance, the *LocationManager* hook in Listing 6.1 (line 2) allows the module to edit or replace the Location object that is returned to the app that requested the current device location.

6.6.2.3 Application Layer

At the application layer, our Android Security Framework provides a mechanism to inject access control hooks into apps themselves. This access control technique is based on the concept of *inlined reference monitors* (IRM) pioneered by Erlingsson and Schneider [49, 48]. The basic idea is to rewrite an untrusted app such that the reference monitor is directly embedded into the app itself, yielding a “self-monitoring” app. Although using IRMs might seem counter-intuitive or redundant in our design, IRMs are the only way in Android’s current app model to achieve privilege separation between the components within an app (e.g., ad libs [84]) or to enforce mandatory results automata policies on file system and network interfaces (e.g., HTTPS-everywhere). The former are DVM internal operations and the latter do not involve the middleware, but instead the app processes interact directly with the file system and network API of the kernel, whose semantics are rather unsuitable for enforcing mandatory results automata policies. Thus, until this app model has been retrofitted to enable a system-centric solution for such kind of policies, our design relies on IRM. ASF provides an instrumentation API that enables security modules to dynamically hook any Java function within an app’s DVM. Hooked functions divert the control flow of the program to the reference monitor, which thereby not only gains access to all function arguments but can also modify or replace the function’s return value. Furthermore, in contrast to the hooks placed in the Android middleware, application layer hooks are dynamic: Hooks are injected by directly modifying the target app’s DVM memory when a new app process is started. This design enables security modules to dynamically create and remove hooks at runtime as well as to inject app-specific hooks.

6.6.3 Middleware Framework API

We elaborate now in more detail on our framework API and the interaction between modules and the security infrastructure. Since we use the existing LSM framework as is, we focus here on our newly introduced middleware security framework and refer to the kernel documentation [112] for details on the LSM API. The middleware framework API of our current implementation contains 168 callback functions. This API can be broken down into the following categories:

Enforcement functions. These functions form the bulk of the API and are called by the framework whenever the enforcement hooks in system apps and services are triggered. Table 6.1 provides an overview of the coverage of our current enforcement hooks. Each hook has a corresponding callback function in the module API, which has the same method signature as the hook (cf. Listing 6.1) and which implements the policy decision logic for its hook. Passing arguments by reference or expecting objects as return values allows these functions to implement mandatory results automata logic.

Kernel Sub-Module Interface. To avoid policy-specific interfaces for the communication between middleware/application layer apps and the Kernel Sub-Module, we introduce a generic kernel module API as part of the middleware framework API. It allows apps and services a controlled access to Linux security modules. Each security

System App/Service	# Hooks	Example hooks
BroadcastQueue	2	deliverToRegisteredReceiver, processNextBroadcast
ContentProvider	12	insert, update, preQuery, postQuery
ActivityStack	5	startActivity, moveTaskToBack, finishActivity
ActivityManagerService	10	checkComponentPermission, checkUriPermission, checkGrantUriPermission
PackageManagerService	21	getPackageInfo, findPreferredActivity, queryIntentReceivers, getServiceInfo, scanPackage, deletePackage
ActiveServices	5	startService, bindService, getServices
LocationManagerService	21	getProviders, requestLocationUpdates, requestGeofence, reportLocation, setTestProviderLocation
AudioService	5	adjustStreamVolume, setMasterVolume, setRingerMode
TelephonyService	2	call, getNeighboringCells
SMSService	7	getAllMessagesFromIcc, sendData, sendText
WiFiService	23	getScanResults, addOrUpdateNetwork, getConnectionInfo, getWifiServiceManager
ClipboardService	7	getPrimaryClip, setPrimaryClip
PowerManagerService	6	acquireWakeLock, userActivity, reboot
PhoneSubInfo	13	getDeviceId, getIccSerialNumber, getLine1Number, getIsimImpi
	Total: 139	

Table 6.1: Break down of hooked system apps and services.

module can implement this interface and internally translate the API calls to calls on the proprietary channel between the user-space and the Linux security module. Two particular challenges for establishing this interface were the self-contained security checks of the kernel module and the requirement that this interface is already available during system boot. To guarantee security, the kernel module is required to perform policy checks to verify that a user-space process is sufficiently privileged to issue commands to it. Additionally, the kernel module is called before the middleware framework can load any MIDDLEWARE SUB-MODULE, e.g., it can be called by Zygote when spawning new app processes. To solve these challenges, our design avoids an additional layer of indirection (i.e., IPC) for communication with the kernel module and loads the interface implementations via the Java reflection API statically into the application framework when it is bootstrapped. This ensures that the calling processes communicate directly with the kernel module through our generic API and that the kernel module can be called independently of middleware services. Listing 6.2 presents the current interface definition.

Listing 6.2: Interface for Access Control Policy Modules to Linux Security Module.

```java
public interface KMAC {
  public boolean init();
  public boolean isEnabled();
  public boolean isEnforcing();
  public boolean setEnforcing(boolean value);
  public boolean setContext(String path, Bundle context);
  public boolean restoreContext(Bundle context);
  public Bundle getContext(String path);
  public Bundle getPeerContext(FileDescriptor fd); /* wrapper around getsockopt call to LSM */
  public Bundle getCurrentContext();
```

```
    public Bundle getProcessContext(int pid);
    public Bundle getConfig(Bundle args); /* e.g., get list of defined booleans or one specific boolean value */
    public boolean setConfig(Bundle conf); /* e.g., set a boolean value */
    public boolean checkAccess(Bundle args); /* args can be, e.g., quadruple of subject ctx, object ctx, object class, op */

    /* Zygote is statically integrated with the Kernel MAC, thus, each KMACAdaptor must implemented these hooks in ZygoteConnection */
    public boolean security_zygote_applyUidSecurityPolicy(Credentials creds, Bundle peerSecurityContext);
    public boolean security_zygote_applyRlimitSecurityPolicy(Credentials creds, Bundle peerSecurityContext);
    public boolean security_zygote_applyCapabilitiesSecurityPolicy(Credentials creds, Bundle peerSecurityContext);
    public boolean security_zygote_applyInvokeWithSecurityPolicy(Credentials creds, Bundle peerSecurityContext);
    public boolean security_zygote_applySecurityLabelPolicy(Credentials creds, Bundle peerSecurityContext);
}
```

Life-cycle management. Every module must implement functions for life-cycle management, such as initialization or shutdown. This enables the framework to inform the module when the system has reached a state during the boot cycle from which on the module will be called or when the system shuts down. Modules should use these functions, e.g., to initiate their policy engines or to save internal states to persistent storage before the device turns off.

Event notifications. Event notification interfaces are used to propagate important system events to the module. For instance, modules should be immediately informed when an app was successfully installed, replaced, or removed. Although this information is usually propagated via a broadcast Intents, the time gap between package change and broadcast delivery might cause inconsistencies in module states. Hence these events must be delivered synchronously.

Framework Callbacks. The framework provides modules a callback interface for communicating in a more direct manner with system services, such as the *PackageManagerService*, and avoids the need to go through the Android API. This is desirable for policy authors that want to leverage the middleware internal information. Our current callback interface, for instance, includes functions that allow modules to efficiently resolve PIDs to application package names.

Proprietary protocols. We introduced in our framework API a *callModule()* function that allows modules to implement proprietary communication protocols with other apps that are aware of this specific module, e.g., the front-end apps (cf. Section 6.6.1). When using *callModule()*, these protocols are based on *Bundles* and enable a protocol similar to the Parcel-based Binder IPC: apps serialize function arguments to a Bundle and add an identifier for the proprietary function the module should execute with the deserialized arguments. It is the task of the module to verify that the sender is sufficiently privileged to send commands.

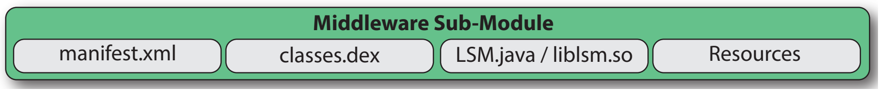

Figure 6.2: Middleware security module structure.

IRM Instrumentation. The framework provides an instrumentation API that enables security modules to hook any Java function within selected app processes. To the best of our knowledge, ASF is the first solution for Android to provide a generic instrumentation API. Hooks injected via the instrumentation API are local to the app process that the API is called from. Therefore, all calls to the instrumentation API need to be performed from within a target app's process. We solve this by placing an instrumentation hook in the *ActivityManagerService* that is triggered when a new app process is about to be launched. A module that implements this hook has to return a Java class for the instrumentation logic that will be executed within the app's process. To ensure that this code is executed before control flow is passed to the app itself, we modify the arguments passed to Zygote to start this new app process via a special wrapper class that loads and executes the instrumentation code first.

6.6.4 Middleware Security Modules

We elaborate in more detail on the structure of security modules. Again, we use Linux security modules as is [112] and, thus, focus here on the MIDDLEWARE SUB-MODULE. A middleware module is simply a *Jar* file that is created with an Android SDK that includes our new security API. It is deployed to a protected location on the file system, from where it is loaded during boot. This Jar file contains all the module's code, resources, and manifest file (cf. Figure 6.2):

Module Manifest. The manifest (formatted in XML) declares properties such as the module author or code version, and, more importantly, the name of the main Java class that forms the entry point for the module.

Classes.dex. The *classes.dex* file contains, as in regular Android apps, the Java code compiled to *Dalvik executable bytecode* (DEX). It contains all Java classes that implement the security module's logic. During the load process of the MIDDLEWARE SUB-MODULE, the middleware framework uses the Java reflection API to load the module's main class (as specified in the manifest) from *classes.dex*. To ensure that the reflection works error-free, the main class must implement the API as described in Section 6.6.3. Since the API defines currently more than a hundred methods, but a security module very likely requires only a subset of those, our SDK provides an abstract class that implements the API. That abstract class can be sub-classed by the module's main class, which then only needs to override the required functions. The abstract class returns for each non-overridden enforcement function an allow decision.

LSM interface. The proprietary interface between the user-space processes and the Linux security module in the kernel is implemented through a native library *liblsm.so* and a corresponding Java class *LSM.java*, which exposes the native library via the Java Native Interface. *LSM.java* has to implement the generic interface for the communication with the kernel that was explained in the previous section. The generic kernel module interface of ASF loads *LSM.java* through the Java reflection API into Android's application framework. This allows apps and services to communicate with the kernel module and avoids a policy-specific interface. We exemplified this mechanism by integrating SELinux through API into Zygote (cf. Section 6.7.4).

Resources. Each module can ship with proprietary resources, such as initial configuration files or required binaries. During module instantiation, the framework informs the module about the filesystem location of its Jar file, enabling the module to extract these resources on-demand from it.

6.6.5 Stackable and Dynamic Loadable Modules

Finally, two desirable properties for implementing an extensible security framework such as our ASF are dynamically loadable policies and policy composition (i.e., stacking modules). In the following we explain why we chose, in contrast to closest related work [90], to *permit* these features by design, but *not* consider them a requirement for our solution.

Dynamically Loadable Modules. Being able to dynamically load and unload modules is desirable, for instance, to speed up the development and testing cycles of modules and, in fact, we used this feature during the development of our example use-cases (see Section 6.7). However, the arguments to support dynamically loadable modules beyond development (e.g., for security add-ons [90]) are disputed: First, dynamic loading is not always technically possible. A small set of static policy models, such as type enforcement [175, P3], require that all subjects and objects are labeled with a security context. Supporting such extensive labeling operations at runtime is an intricate problem. Second, there exist security considerations. The loading and unloading of modules must be strictly controlled to ensure that only integrity protected, trusted modules are loaded. Otherwise, given the privileges of modules, this would open the way to powerful malware modules. In our design we agree with the conclusions of the various Linux security module authors [40] and consider the drawbacks of dynamically loadable modules to outweigh their benefits. Therefore, we load the module *once during the system boot* and *permit* users of our framework to additionally activate dynamic unloading and loading of modules. But we currently do not consider this feature a requirement for our solution.

Stackable Modules. Composing the overall policy from multiple, simultaneously loaded and independent policies is a desirable feature, since usually no "one-size-fits-all" policy exists. Android, for instance, implements currently a quadruple-policy approach consisting of Permissions, SE Android type enforcement, AppOps, and Linux capabilities—each being responsible for a different aspect of the overall access control strategy. Multiple policies will naturally conflict and thus require the security framework

Existing solution	**LoC of module**	**LoC added/removed/edited (total delta)**
AppGuard [7]	5,059	+828/-79/∘13 (18.18%)
CRePE [24]	3,682	+915/-48/∘45 (27.38%)
XManDroid [P5]	3,244	+153/-14/∘28 (6.01%)
AppOps / Intent Firewall	2,290	+627/-106/∘39 (33.71%)
FlaskDroid [P3]	4,968	+749/-32/∘40 (16.53%)

Table 6.2: Effort of porting different security extensions as module on our ANDROID SECURITY FRAMEWORK.

to support different policy composition and reconciliation strategies (e.g., consensus or priority based) [152, 120]. However, supporting fully generic policy composition is quite a challenge and has been shown to be intractable [67]. Thus, despite its benefits, we decided in our design to follow the lessons learned by the LSM developers [206] and to only *permit* module developers to implement stackable modules, but we do not provide explicit interfaces for stacked modules in our framework infrastructure. In module combinations where policy consolidation is known to be feasible, the approach to stacking modules would be to provide a “composition module” that implements policy reconciliation and composition logic and which in turn can load other modules and multiplex API calls between them.

6.7 Example Security Modules

In this section, we demonstrate the efficiency and effectiveness of our ANDROID SECURITY FRAMEWORK by instantiating different security models from related work. To illustrate the versatility of ASF, we chose models from the areas of inlined reference monitoring, context-based access control, domain isolation, and type enforcement.

6.7.1 AppOps and IntentFirewall

Google introduced (unofficially) with Android v4.3 the *AppOps* infrastructure for dynamic, more fine-grained Permissions. It added hooks in different system services and apps, which query a central AppOpsService whether an application is allowed to perform an operation (e.g., retrieving the location of the device or querying a *ContentProvider*). The AppOps rules define a mapping from UID/package name to allowed operations. AppOps offers an interface to apps to retrieve the current configuration. Additionally, Google introduced (again unofficially) an *IntentFirewall*, which acts as a reference monitor for certain Intent-based operations like starting an Activity. The IntentFirewall rules describe which caller is allowed to receive which kind of Intent object, using the Intent’s attributes such as destination component. The sending or processing of Intents that violate these rules is aborted.

Implementation as a module We ported AppOps and IntentFirewall (from Android v4.3) to a security module for ANDROID SECURITY FRAMEWORK (cf. Figure 6.3) by moving the AppOpsService and the IntentFirewall classes into a module. Our module comprises 2,290 lines of code and differs in 33.71% of all LoC from the original

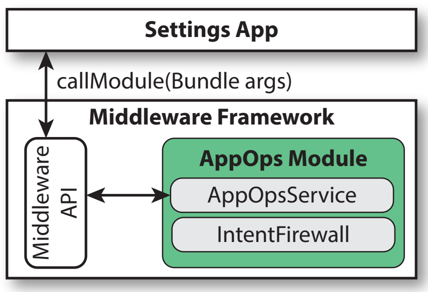

Figure 6.3: AppOps and IntentFirewall security module.

implementation. The bulk of the changes (520 LoC), were required to move the hook logic of both services from the system apps and services of Android into the module by using our enforcement functions. For the IntentFirewall, this was straightforward and we only had to substitute a direct callback from IntentFirewall to the *ActivityManagerService* by our framework callback mechanism. For the AppOpsService, we had to add a mapping from caller PID to package name. By default the hooks of AppOps determine the caller's package name and pass this information to the AppOpsService for policy check. Since this is a policy-specific logic of the hooks, our framework hooks do not (by default) provide the caller's package name and we re-implemented this logic in our module by using our callback interface, which allows us to retrieve the package name for app PIDs. Moreover, we adapted the AppOpsService interface to retrieve/configure the current policies via a Bundle-based communication. AppOps is, furthermore, partially integrated into the *Settings* application to allow users to disable notifications from selected apps. We replaced this policy-specific channel between Settings and AppOps also with our policy-agnostic Bundle-based communication. Modules that support this Settings option, can return a value indicating whether notifications are disabled or not. If the module does not support this feature, the Settings app by default allows notifications. However, our AppOps module does currently not support the operation watching feature, which requires the registration of application callback objects with the module.

6.7.2 CRePE

CRePE [24] is a security extension to Android v2.3 that enforces fine-grained and context-related access control policies. The context is based on the geolocation of the device and, depending on this location, CRePE either allows or denies apps access to security and privacy sensitive information. The security policies can be deployed over different channels, e.g., via SMS. To enforce the policies, CRePE hooked all relevant system services, e.g., to override Android's default permission check with its context-related check.

Implementation as a module. We ported CRePE[1] as a security module for ASF by moving its policy engine class *CRePEPolicyManagerService* and related classes, which

[1]Source code retrieved from `http://sourceforge.net/projects/crepedroid`

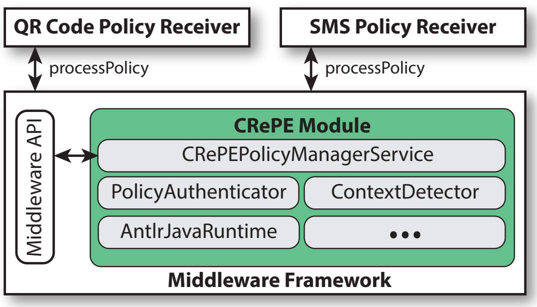

Figure 6.4: CRePE security module.

were originally running as a separate system service, into a module (cf. Figure 6.4). On initialization, CRePE's context detector registers as a listener for location updates to detect context changes. Additionally, we used the enforcement functions of our API to re-implement the logic of CRePE's hooks. Furthermore, CRePE uses front-end apps to parse and inject policies from different channels. We moved the policy parser into the module and established a Bundle-based communication protocol between the front-end apps and the module to forward received policies for processing. We used the example policies shipped with the CRePE source code to successfully confirm that the enforcement by our module yields the same results as the original CRePE implementation.

Our port of CRePE as a security module consists of 3,682 lines of code (cf. Table 6.2), excluding the unmodified ANTLR runtime (7,526 LoC). Of these 3,682 LoC 27.38% were changed during the port. The bulk of this difference, 817 LoC, is attributed to relocating the policy parser. Implementing the Bundle-based communication protocol added 74 LoC. Only 2 LoC had to be changed to adapt CRePE's calls to the Android API from its original Android v2.3 implementation to our Android v4.3 implementation.

6.7.3 XManDroid

XManDroid [P5] (see also Chapter 4) extends the security architecture of Android v2.2.1 to enforce Chinese Wall policies between apps that might jointly leak privacy sensitive information. It uses hooks within different system services in the middleware and TOMOYO Linux at the kernel level to monitor all access control requests, reflect these interactions between processes/apps in a graph model, and use this model to check against policies whether an inter-app communication would lead to an attack state. If so, it denies the new communication. The policy decision logic is implemented as an extension to the *ActivityManagerService*.

Implementation as a module. We ported our XManDroid to a module for ASF (cf. Figure 6.5) by extracting the policy decision logic from the *ActivityManagerService* and moving it into a module. Using the enforcement functions of our API we moved the XManDroid hook logic to this module as well and, by using a proprietary channel, we enabled the middleware extension to communicate from the module with the TOMOYO kernel module. The kernel was specifically compiled and deployed with

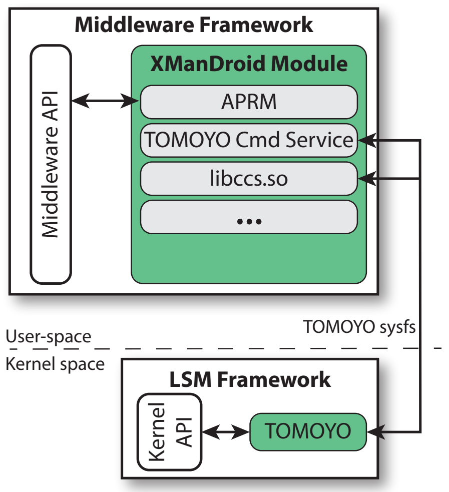

Figure 6.5: XManDroid security module.

a TOMOYO Linux security module. The XMANDROID source code comes with an example configuration for the policy described in [P5] and we used this configuration to successfully confirm that our module yields the same enforcement results as the original XMANDROID implementation.

Our XMANDROID middleware module consists of 3,244 lines of code, excluding the unchanged JGrapht library (9,256 LoC). Our module differs in only 6.01% (195 LoC) from the original implementation. Of these 195 LoC, 141 are attributed to additions necessary for porting XMANDROID's filtering logic for broadcasts from the *ActivityManagerService* to the module.

6.7.4 Type Enforcement

SE Android [175] brought SELinux type enforcement to the Android kernel and established the required user space support, e.g., it extended Zygote to label new app processes with a security type. Our FLASKDROID [P3] (see also Chapter 5), developed for Android v4.0.3, extends SE Android's type enforcement to Android's middleware. Building on SEAndroid's kernel and low-level patches, it adds policy-specific hooks as policy enforcement points to various system services and apps in Android's middleware. The policy decisions at kernel level are made by the SELinux kernel module, while the decisions at middleware are made centrally in a policy server service. Both policy decision points decide based on subject type, object type, and object class reported by the hooks at their respective layer whether control flows should be truncated or not.

Implementation as module. We realized type enforcement with our ASF by porting FLASKDROID[2] as a module (cf. Figure 6.6). At kernel level, we use the SE Android kernel and provide an SELinux-specific interface implementation for the kernel module, in this context porting the currently hardcoded SELinux support of Android's middleware into

[2]Source code retrieved from `http://www.flaskdroid.org/`

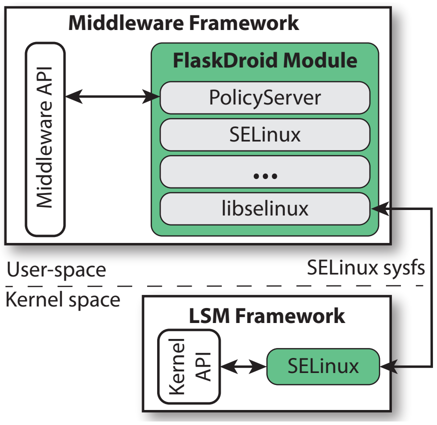

Figure 6.6: FlaskDroid module.

an ASF module. A technical description of this interface implementation is provided in the following Section 6.7.4.1. Further, we moved the middleware policy server and its dependencies into the middleware module. Using the enforcement functions of our API, we moved the policy-specific hook logic of FlaskDroid into the module as well. More technical details on moving the hook logic are provide in the following Section 6.7.4.2). Additionally, we used SE Android's build system to label the file system with security types.

Our port of FlaskDroid's middleware component as a security module consists of 4,968 lines of code (cf. Table 6.2) and differs in only 16.53% of all LoC from the original code. The bulk of these changes (550 LoC) is attributed to additions for implementing a mapping from the enforcement functions of our framework API to FlaskDroid's type checks. To confirm the correct enforcement of policies, we used the policies for middleware and kernel level that are provided with the FlaskDroid source code. Additionally, we noticed during our tests that the original implementation contains an error in assigning middlware security types to processes: It maps process UIDs always to the security type of the first package in a shared sandbox, although the policy can define different types for packages that share a UID. Additional changes were necessary to fix this error in our FlaskDroid module.

6.7.4.1 Policy-agnostic Calls from Zygote to Linux Security Module

We briefly explain at the example of Zygote how the generic interface for calls to the kernel module can be used. As described in our architecture section (Section 6.6), we added a generic interface implementation, called `KMAC.java`, to the Android API. `KMAC.java` implements the interface described in Listing 6.2 at page 120. `KMAC.java` in turn loads via the Java reflection API the `LSM.java` classes and via JNI the `liblsm.so` deployed by modules and uses them to forward calls to the kernel module. `LSM.java` must hence also implement the interface in Listing 6.2. `LSM.java` and `liblsm.so` are responsible for translating the function arguments to the kernel module specific protocol. For instance, consider Listing 6.3 that shows how the `KMAC` interface is used in Zygote to verify that the caller is allowed to specify certain parameters like the UID/GID of a new app process. It first uses the `getPeerContext` function to retrieve the kernel-level security context of the calling process. This information is stored in a generic Bundle structure. It afterwards uses this information to request policy decisions from the Linux security module such as `security_zygote_applyUidSecurityPolicy` (a Zygote-specific hook). Listing 6.4 shows how the SE Android module implements the interface to translate the arguments to SELinux-specific arguments and to call the SELinux kernel module. Here, `SELinux.java` takes the role of `LSM.java` and `SELinux.getPeerContext` and `SELinux.checkSELinuxAccess` are *native* functions that call via `libselinux.so` the kernel module. Hence, `libselinux.so` takes the role of `liblsm.so`.

Listing 6.3: Use of generic Kernel module interface in `ZygoteConnection.java`.

```java
private final Bundle peerSecurityContext;
private static final KMAC mKMAC = new KMAC();
...
ZygoteConnection(LocalSocket socket) throws IOException {
...
  peerSecurityContext = mKMAC.getPeerContext(mSocket.getFileDescriptor());
...
}
...
private static void applyUidSecurityPolicy(Arguments args, Credentials peer
    , Bundle peerSecurityContext) {
...
  boolean allowed = mKMAC.security_zygote_applyUidSecurityPolicy(peer,
      peerSecurityContext);
...
}
```

Listing 6.4: Implementation of the generic LSM interface for SELinux kernel module.

```java
package android.os;

public class SELinuxAdaptor implements KMACAdaptor {
...
  @Override
  public Bundle getPeerContext(FileDescriptor fd) {
    String ctx = SELinux.getPeerContext(fd);
    Bundle ret = new Bundle();
    ret.putString("selinux.context", ctx);
    return ret;
  }

  @Override
  public boolean security_zygote_applyUidSecurityPolicy(Credentials creds,
      Bundle peerSecurityContext) {
    String peerCtx = peerSecurityContext.getString("selinux.context");
    return SELinux.checkSELinuxAccess(peerCtx, peerCtx, "zygote", "
        specifyids");
  }
...
}
```

6.7.4.2 Details on FlaskDroid Hook Logic

We illustrate at the example of FlaskDroid [P3] how the hook logic of existing solutions can be moved into a module. Listing 6.5 shows one of the original FlaskDroid hooks in the `getAllProviders` function of the Android location service. The hook calls via the service's context to FlaskDroid's policy server where the access control decision is determined by the `checkPolicy` function. This function internally determines the subject's and object's security type from their UIDs.[3] A denial of access results always in a security exception that is thrown back to the caller of the location service API. Listing 6.6 shows the re-implementation of this logic in a module for our ASF by simply overriding the corresponding enforcement function for `LocationManagerService.getAllProviders` and directly calling the `checkPolicy` function with all required parameters provided by the hook. It should be noted that in FlaskDroid the security exception in case of denied access is hardcoded within the system, while as an implementation as a module the FlaskDroid module could alternatively change the enforcement to a less interruptive enforcement by reassigning the `providerList` parameter to an empty list of Strings, i.e., pretending to the calling app that there is no location provider present in the system. In fact, as a module, such a change in strategy can be more easily rolled out than as a hardcoded implementation within the middleware.

Listing 6.5: Original FlaskDroid hook in `com.android.server.LocationManagerSe`

```java
public List<String> getAllProviders() {
...
  if(mContext.checkSecurityContext(Binder.getCallingUid(), Process.myUid(), "locationService_c", "getAllProviders") != PackageManager.PERMISSION_GRANTED) {
    throw new SecurityException("Denied by MAC policy");
  }
...
```

Listing 6.6: Re-implementation of the hook from Listing 6.5 in a security module.

```java
@Override
public void security_location_getAllProviders(List<String> providerList, int uid, int pid) {
  if(checkPolicy(uid, Binder.getCallingUid(), "locationService_c", "getAllProviders") == PackageManager.PERMISSION_DENIED) {
    throw new SecurityException("Denied by MAC policy");
  }
}
```

6.7.5 Inlined Reference Monitoring

We use AppGuard [7] as the use-case to illustrate the applicability of our IRM instrumentation API, but similar application rewriting approaches [101] are also feasible. AppGuard is a privacy app for Android that enables end-users to enforce fine-grained access control policies on 3rd party apps by restricting their ability to access critical

[3] This is the original FlaskDroid behavior that is, as mentioned earlier, flawed.

system resources. By inlining reference monitors into application code, this approach supports security policies not easily enforceable by traditional system-centric reference monitors in the Android middleware or kernel, e.g., to enforce the use of *https* over *http*.

Implementation as a module. We ported AppGuard as a module for ASF by splitting its original app into three components: We adapted the (1) AppGuard reference monitor with its dynamic hook placement and policy enforcement logic to use our IRM instrumentation API. The reference monitor is injected into selected app processes via our framework at app startup. The policy decision logic and persistent storage of policy settings was moved into (2) a middleware module. The middlware module selects the apps into which the IRM is injected. It also implements a Bundle-based communication protocol to exchange policy decisions and security events with the IRM component and with (3) a front-end app. The front-end app allows the user to adjust policy settings and to view logs of security-relevant events. We used the policies included in the original AppGuard implementation to confirm that policy enforcement by our module and by the original implementation are identical.

Our AppGuard security module consists of 5,059 LoC in total (cf. Table 6.2), with 782 LoC residing in the middleware module and 4,277 LoC in the IRM. Our module diverts in 18.18% of all LoC from the original code. The majority of the differences, 728 LoC, is attributed to moving the policy decision logic into the middleware module, while only 46 LoC were required to adapt the inlined reference monitor to use the provided instrumentation API.

6.7.6 Saint

Saint [140] is an extension for Android OS v1.5 that allows app developers to ship their apps with policy rules that determine how the app can interact with other apps in the system. For instance, the rules can declare that only apps with a specific package name, version, or set of permissions are allowed to call the app or be called by the app. The rules also support defining Intent attributes as rule criteria. The rules are enforced by the system through hooks in different system apps and services, such as the *ActivityManagerService*, that allow monitoring operations for the different app component types. Additionally, Saint provides a front-end app (*FrameworkPolicyManager*), that allows the user to override developer policies.

Implementation as a module. We *re-implemented* Saint as a security module by developing a module (729 LoC) that supports Saint's policy language as described in [140]. We use our event functions to extract policy files from newly installed application packages and insert them into a policy database in our module. We use different hooks in the *ActivityManagerService* (e.g., starting an Activity, resolving an Activity, finding active services), Broadcast subsystem, or *ContentProvider* class to enforce the Saint *runtime* policies. Using the `scanPackage` hook in the *PackageManagerService* we enforce Saint *install-time* policies to decided whether a new app is installed. Communication between the module the front-end app is again implemented based on Bundles. We successfully verified our Saint module's effectiveness using the policies for the running example described by the Saint author's [140] and a set of test apps that implement Saint's example scenario.

6.7.7 TrustDroid

TrustDroid [P4] (see also Chapter 3) extends the Android OS v2.2 architecture with isolation of different domains such as "work" and "private". Every application is classified during installation into one of the available domains. For classification, TrustDroid uses package-specific attributes such as developer signature, external signature, or package name. Enforcement hooks at middleware and kernel level prevent at runtime any communication between different domains. At kernel level, TrustDroid uses TOMOYO Linux to enforce the policies. Policy rules for newly classified apps are propagated from the middleware to the kernel.

Implementation as a module To *re-implement* TrustDroid as a security module (862 lines of code), we deployed a TOMOYO-enabled Linux kernel on the device (i.e., our Kernel Sub-Module) and developed a Middleware Sub-Module that deploys the required `LSM.java` and `libccs.so` to communicate with the kernel module. Additionally, we used our `scanPackage` hook in the *PackageManagerService* to classify newly installed applications and keep a mapping from UID to domain.[4] Because TrustDroid's policy is static and very simple, its architecture does not distinguish between policy enforcement and policy decision points, but instead every hook retrieves the domain of the current subject and object and denies access if their domains differ. We re-implemented this logic using the enforcement functions of our module, which was a straightforward implementation. For *ContentProviders* (e.g., Contacts) TrustDroid classifies the database entries and returns on access only the entries that have the same domain as the caller. Since we prohibit by design such policy-specific intrusions into the default *ContentProviders*, we use our pre-query hooks to modify selection arguments to retrieve only contacts that are allowed for the current caller (e.g., where the contact's group indicates a private contact). Using two example applications that are classified differently, we verified the effectiveness of our TrustDroid module.

6.7.8 Data shadowing

Both *AppFence* [93] as well as *TISSA* [218] provide a data shadowing feature. Data shadowing means, that an application that wants to retrieve sensitive information (e.g., contacts information, IMEI number, or location data) only receives empty, fake, or filtered data.

Implementation as a module. We *re-implemented* the data shadowing features of AppFence and TISSA as a module by using our *mandatory results automaton* hooks in the `ContentProvider.Transport` class, the *ContactsProvider*-specific hooks, Telephony service and Location service. For *ContentProvider* and *ContactsProvider*,

[4]TrustDroid does not allow apps in a shared sandbox to be classified differently.

	Frequency	**Mean** (μs)
Stock Android 4.3	7,320	116.182±4.550
ASF v4.3	6,009	129.851±5.681

Table 6.3: Weighted average performance overhead of executing hooked functions in stock Android and in our ANDROID SECURITY FRAMEWORK. The margin of error is given for the 95% confidence interval.

in particular our pre-query and post-query hooks allowed us a fine-grained filtering or replacing (faking) of the returned data as well as returning an empty data set. However, the current coverage of our enforcement hooks does not include some of the data shadowing points of AppFence, such as microphone, logs, or camera, and we plan on adding them in the future.

6.7.9 Kirin

Kirin [44] extends Android's application installation process with policy-based checks and denies installation of a new app when it violates the policy. Based on its time of publication, we presume that it was developed for Android OS v1.5.[5] The actual policy check was performed in a dedicated Android application developed for Kirin, which interacted with the installation process. These policies are based on the set of permissions requested by an app and the interfaces (e.g., Broadcast receivers) it wants to register in the system. The installation of apps that are rejected by the policy is denied.

Implementation as a module. To *re-implement* Kirin's security service as a security module, we developed a module that supports Kirin's security language. Using our `scanPackage` hook in the *PackageManagerService*, we check new applications against the policy and abort their installation in case the policy rejects the application. Our Kirin module comprises 246 lines of code.

6.8 Evaluation

In this section we evaluate the performance of our ANDROID SECURITY FRAMEWORK and discuss its current scope and prospective future work.

6.8.1 Performance

Although the actual performance overhead strongly depends on the overhead imposed by the loaded module, we wanted to establish a baseline for the impact of our ANDROID SECURITY FRAMEWORK on the system performance. The performance of LSM has been evaluated separately, e.g., for SEAndroid [175], and we are interested here in the effect of our new middleware security framework on the performance of instrumented middleware system services and apps.

[5] `http://en.wikipedia.org/wiki/Android_version_history`

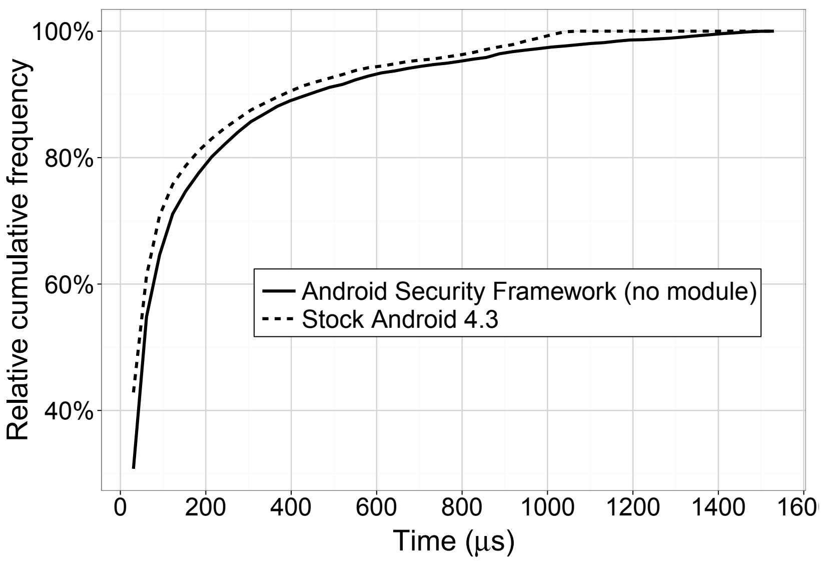

Figure 6.7: Relative cumulative frequency distribution of micro benchmarks in stock Android (dashed line) vs. ANDROID SECURITY FRAMEWORK (solid line).

Methodology. We implemented our ASF as a modification to the Android OS code base in version 4.3_r3.1 (*"Jelly Bean"*) and used the Android Linux kernel in branch *android-omap-tuna-3.0-jb-mr1.1*. We performed micro-benchmarks for all execution paths on which a hook diverts the control flow to our middleware framework: We first measured the execution time of each hooked function with no security module loaded and allowing all access. Afterwards we repeated this test with hooks disabled to measure the default performance of the same functions and thus operating like a stock Android. All our micro-benchmarks were performed on a standard Nexus 7 development tablet (Quad-core 1.51 GHz CPU and 2GB DDR3L RAM), which we booted and then used according to a testplan for different daily tasks such as browsing the Internet, sending text messages and e-mails, contacts management, or (un-)installing 3rd party apps.

Micro-benchmark results. Table 6.3 presents the number of measurements for each test case and their mean values. To eliminate extreme outliers, we excluded in both measurement series the highest and lowest decile of the measurements. For ASF the mean is the weighted mean value with consideration of the frequency of each single hook. Table 6.4 provides a break down of the most frequently called hooked Android API functions and their mean execution time. In overall, our framework with no loaded module imposed with 129.851 μs approximately only 11.8% overhead compared to stock Android. Figure 6.7 presents the relative cumulative frequency distribution of our measurements series and further illustrates this low performance overhead.

Module Performances. Table 6.5 provides an overview of the performance impact of selected security models as reported in their respective publications (*native*) and as measured for their implementation as ASF module (ASF *Module*). However, it should

	Android Security Framework		Stock Android v4.3	
Hooked function	**Frequency**	**Mean (μs)**	**Frequency**	**Mean (μs)**
AMS.checkComponentPermission	1,705	39.413±0.658	2,024	36.518±0.523
BroadcastQueue.broadcastIntent	908	305.274±16.752	1,007	332.328±17.085
SettingsProvider.call	544	67.710±3.004	669	46.574±1.723
PaMS.queryIntentReceivers	438	92.458±3.598	745	84.343±2.296
PaMS.queryIntentActivities	296	192.178±15.458	242	195.211±18.355
PoMS.acquireWakeLock	229	296.246±10.740	255	295.601±11.121
PaMS.getActivityInfo	229	53.039±2.223	203	45.551±2.104
PaMS.getPackageInfo	207	47.324±2.339	307	37.774±1.456
PaMS.queryIntentServices	123	131.744±9.220	134	106.354±6.069
PaMS.getPackageUid	93	35.767±2.353	201	30.005±0.000

AMS: ActivityManagerService ; PaMS: PackageManagerService; PoMS: PowerManagerService

Table 6.4: Ten most frequently invoked hooked functions and their average performance overhead on ASF vs. stock Android v4.3. The margins of error are given for the 95% confidence interval.

Use-case	Implementation	Android	Test device	Average (μs)
CRePE [24]*	Native	v2.3	HTC Magic	≈ 100
	ASF Module ‡	v4.3	Nexus 7	168.943±5.884
XManDroid [P5]	Native $^\diamond$	v2.2	Nexus One	532
	ASF Module ‡	v4.3	Nexus 7	206.062±5.573
FlaskDroid [P3] (middleware)†	Native	v4.0.3	Galaxy Nexus	452
	ASF Module ‡	v4.3	Nexus 7	359.317±11.015

Table 6.5: Performance measurements of our example modules.
* Two rules loaded. $^\diamond$ Weighted average for un-/cached checks. † With basic policy loaded.
‡ Weighted average incl. IPC roundtrip between hook and module.

be noted, that these measurements are not directly comparable, because all security models have originally been implemented for a different Android OS version and been tested on a different hardware platform. Figure 6.8 presents the cumulative frequency distribution for the measured performance overhead of our example modules versus stock Android v4.3.

6.8.2 Current Scope and Future Work

System setup. Certain security models require a preparatory system setup. For instance, type enforcement requires a pre-labelling of all subjects and objects. After the system has been setup, ASF supports modularization of these security models (cf. Section 6.7.4).

Module Integrity. As part of the kernel, the KERNEL SUB-MODULE has the highest level of integrity and additional means [6, 63] to protect the kernel integrity could be deployed. In contrast, the MIDDLEWARE SUB-MODULE, as a user space process, can be circumvented or compromised by attacks against the underlying system (e.g., root exploits) and thus requires support by the kernel modules to prevent low-level privilege escalation attacks. Inlined reference monitors are inherently susceptible to attacks by malicious applications [86], because the reference monitor executes in the same process

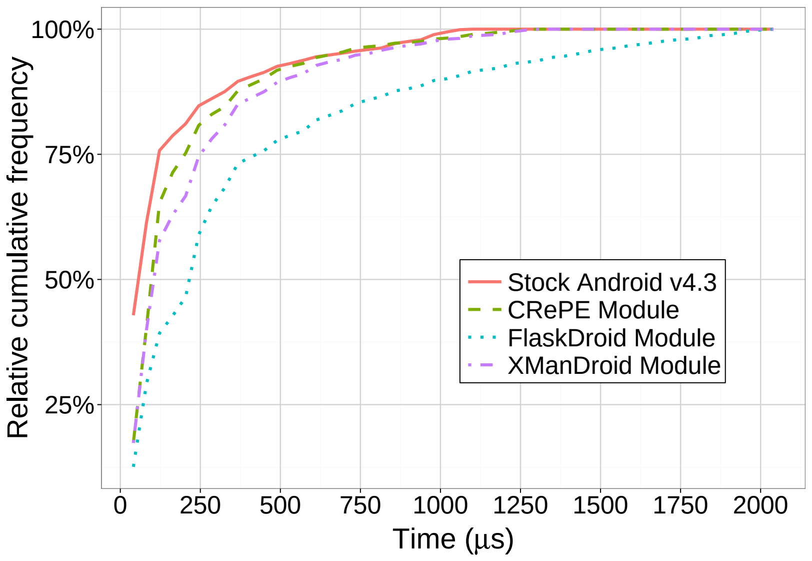

Figure 6.8: Relative cumulative frequency distribution of example modules' performance overhead vs. stock Android v4.3.

as the application that it monitors and no strong security boundary exists between the monitor and the app code. To remedy this situation, we are currently retrofitting Android's application model to combine the benefits of inlined and of system-centric reference monitors. By splitting apps into smaller units of trust (e.g., app components and ad libs), system-centric reference monitors are able to differentiate distinct trust levels within apps [151, 196, 170].

Future work will investigate how Android's application model can be retrofitted to put our ASF into a sweet spot that combines the benefits of inlined reference monitors and system-centric reference monitors. By splitting apps into smaller units of trust, e.g., native code, app components, or external libraries (e.g., [170, 144]), system-centric reference monitors are able to differentiate distinct trust levels within apps. This enables a more fine-grained and effective enforcement [151] similar to IRMs but with a higher level of integrity.

Completeness. It is crucial for the effectiveness of our security framework, that *all* access to security and privacy sensitive resources is mediated by the reference monitors. We consider it out of scope for this thesis to formally verify the completeness of our prototype framework, but plan to use recent advances in static and dynamic analysis on Android to verify the placement of our hooks, similarly to how it was done for the LSM framework [41, 60]. In particular, this requires further investigation of the extent the API-oriented design and convergence of privileged functionality into designated system services and apps [212, P3] aids the verification process.

Information flow control. Our framework provides modules with the control over which subject (e.g., app) has *access to* which objects (e.g., device location), but it cannot control how privileged subjects *distribute* this information. Controlling information flows is an orthogonal problem specifically addressed by different solutions [46, 156]. We plan to integrate such data flow solutions into our framework and to extend our security API with new generic calls for taint labeling and taint checking.

6.9 Conclusion

In this work we presented the Android Security Framework (ASF), an extensible and policy-agnostic security infrastructure for Android. ASF allows security experts to develop Android security extensions against a novel Android security API and to deploy their solutions in form of modules or "security apps." Modularizing security extensions overcomes the current unsatisfactory situation that policy authors are either limited to one predetermined security model that is embedded in the Android software stack or that they are forced to confide in a security-model-specific Android fork instead of the mainline Android code base. Additionally, this modularization provides a number of benefits such as easier maintenance and direct comparison of security extensions. We demonstrated the effectiveness and efficiency of ASF by porting different security models from related work to ASF modules and by establishing a baseline for the impact of our infrastructure on the system performance.

7 Scippa

System-Centric IPC Provenance on Android

7.1 Motivation

Smartphone operating systems allow end-user customization of the phone's functionality with 3^{rd} party apps. To make this extensibility possible and to simultaneously protect the end-user's privacy, current designs of smartphone operating systems exhibit a complex combination of sandboxing-based privilege-separation and extensive message-based data-sharing. Android facilitates the integration of remote services and data into an app using a very lightweight IPC mechanism called *Binder*, which forms the primary channel for inter-app communication. To realize privilege-separation between apps, Android implements app sandboxing by assigning each app a distinct user ID under which the app's processes are executed. To implement the least-privilege principle, *Permissions*, i.e., privileges, are assigned to UIDs. Android ships with a set of pre-defined permissions to protect the Android application framework API, for example, reading the user's address book or retrieving the device's geolocation. It further allows app developers to define custom permissions to protect their apps' interfaces. Protecting an app interface with permissions is realized by (1) statically declaring in a manifest file which permissions are required from a caller to successfully access each app component; or by (2) performing runtime checks in the app components using IPC provenance information provided by Binder, i.e., the calling process' UID.

7.2 Problem Description

However, Android's current solution to protecting app interfaces is unsatisfactory. Statically declaring the required (custom) permissions for interacting with app components reduces the scope of access checks to the permissions each app holds and excludes more app-specific information such as the developer ID or package name. This solution is not scalable and makes it virtually impossible to flexibly endorse specific apps for component access: permissions can be either requested by any app or require apps to be signed with the same key, which in turn would require a more flexible public key infrastructure that does not exist. Performing runtime checks to protect app components, on the other hand, is more flexible and allows more fine-grained access control, but requires that the IPC mechanism provides message provenance information to app components. Android's system model does *not* fulfill this requirement for sufficient IPC provenance information for all app component types. While it provides the caller UID to *ContentProvider* and *Service* components, it fails in providing this information to components that are receivers of Intent messages–the most prominent inter-app communication mechanism. Prior work [148, 21, 30, 44, 36] has identified different attacks that can occur as a result of this shortcoming, most prominently *confused deputy attacks* [88].

7.3 Contributions

In this work, we identified the technical root-cause for this shortcoming in providing IPC provenance information—a mismatch between Android's concept for inter-application communication at its middleware layer and at its kernel layer. In this multi-layered communication framework, the Binder kernel module is responsible for providing the sending process' user ID to the receiving process. However, *logical* communication occurs between app components—in the literature commonly referred to as *Inter Component Communication* (ICC) [44, 45]—using different abstraction levels of Binder IPC at Android's middleware layer. These abstractions introduce indirections and message dispatching that cause Binder's IPC provenance information (i.e., the sender UID) to be lost along the ICC control flow between app components.

We then present Scippa, our extension to Android's inter-application communication framework, to remedy this shortcoming of Android's architecture. Scippa builds Binder IPC call-chains for ICC control flows and thus provides the required provenance information to apps. Although Quire [36] first identified the need for provenance information on Android, Scippa is, to the best of our knowledge, the first approach that *directly* addresses the mismatch between the middleware and the kernel-level security design in Android's multi-layered inter-application communication framework.

At the core of Scippa is an extension to the Binder kernel module, which constructs and forwards IPC call-chains across distinct application processes. The kernel module extension is complemented by extensions to the message handling routines in Android's application libraries to propagate call-chains across all components of an app. In contrast to Quire's prototypical implementation, which requires app developers to explicitly pass on call-chain information during ICC, our extensions integrate seamlessly into the system architecture and call-chains are established transparently to apps and app developers. Only when developers want to retrieve call-chains, they must be aware of a new system API.

Scippa enables for the first time determining the ICC caller ID within all types of Android app components. For instance, apps can now identify the sender of received broadcast Intents and thus distinguish spurious notifications from benign ones; they can also detect if a security-sensitive *Activity* was invoked from a trustworthy caller. This allows apps in general a more fine-grained, flexible self-governing of their interaction with other apps and provides the means to effectively mitigate recently reported attacks such as confused deputy attacks [88, 148, 30]. Additionally, we present and discuss changes to Android's app model to enable the return of finalized call-chains to the sending app. Providing information about how their messages were distributed, both by the system and other apps, gives senders the means to *detect* spurious or malignant distribution of their messages (e.g., message hijacking [21]).

The evaluation of our prototype implementation for Android v4.2.2 shows that the performance overhead imposed on Binder IPC messages is only 2.23% and thus does not impede the overall system performance.

7.4 Binder-Based Inter-App Communication on Android

We first provide necessary technical background for Scippa that complements the general background information on Android that we provided in Chapter 2. We describe in more technical depth how Android uses Binder-based IPC for different types of inter-app communication. In particular, we explain how Binder transactions are integrated into Android's security design and their relevance for access control architecture such as our *FlaskDroid* (see Chapter 5) or *Android Security Framework* (see Chapter 6).

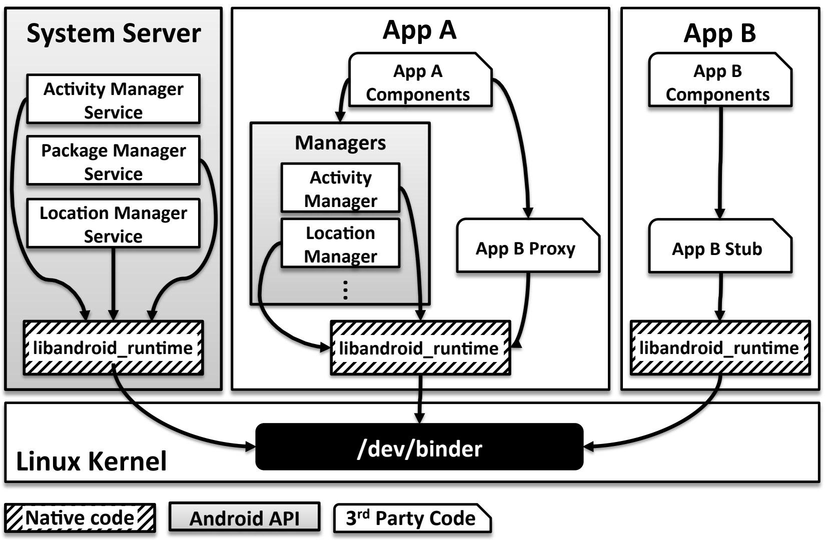

Figure 7.1: Binder-based inter-component communication.

7.4.1 Binder-based ICC

Although Android builds on top of a Linux kernel that provides *"classical"* channels such as files or sockets, the primary IPC mechanism on Android is *Binder*. In the following we take a top-down approach to ICC in Android. We show how Android uses ICC and explain afterwards how ICC is implemented as Binder transactions. We refer to external documentation [162] for more details on Binder.

7.4.1.1 Using Binder-Based ICC on Android

Figure 7.1 provides a high-level overview of standard Binder IPC in Android when used for connecting components of different apps. Apps can, for instance, either contact system services such as the *LocationManager* or communicate directly with each other. All Inter-Component Communication (ICC) builds on top of Binder IPC. User space processes can communicate with each other over Binder IPC via the Binder kernel module that is exposed through the `/dev/binder` sysfs entry. For inter-component communication, the *libandroid_runtime* library, included in all apps, includes an implementation of the Binder communication protocol. Moreover, since application developers usually do not want to deal directly with the low-level mechanics of inter-process communication, Android's design provides different levels of abstraction for Binder IPC. These allow developers to easily make use of Binder IPC at the application level to connect different apps' components (cf. Figure 7.1 for Stubs, Proxies and Managers).

Stubs and Proxies. The most basic level of abstraction of Binder IPC are Stubs and Proxies, which implement remote procedure calls (RPC) via Binder IPC. A Proxy at the caller-side marshals the method parameters into primitive data types and transfers them via IPC to the recipient, where Stub unmarshals the primitives into the original parameters and calls the actual method. This concept of thread execution in which the

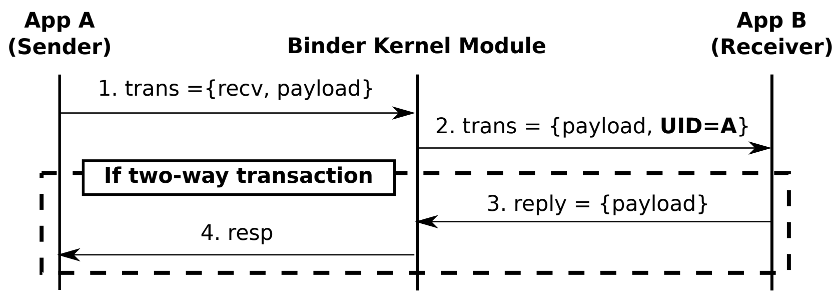
App A
(Sender)
Binder Kernel Module
App B
(Receiver)
1. trans ={recv, payload}
2. trans = {payload, **UID=A**}
If two-way transaction
3. reply = {payload}
4. resp

Figure 7.2: Binder transaction protocol.

logical flow of control temporarily "hops" to a thread in another process is also referred to as *"thread migration"* [55, 143].

System Services and Managers. MANAGERS are part of the SDK and encapsulate pre-compiled PROXIES for system apps and services like the *LocationManager* that implement the Android application framework API. This eliminates the need for compiling PROXIES for the default API during app build-time.

Intents. The highest level of abstraction are so-called *Intent* messages. An Intent is a data structure used to provide an abstract description of an operation to be performed by its receiver(s). Common usages of Intents include starting *Activity* components or broadcasting notifications to apps. Since the sender of an Intent can both explicitly state the target component and implicitly define potential receivers through a description of the intended action, the actual target app(s) must be resolved at runtime. This is the task of the *ActivityManagerService*, which relays all Intents.

7.4.1.2 Binder Transactions and Integration Into Android's Security Design

Before we explain how Binder acts as a building block in Android's security architecture, we first explain how Binder conducts transactions between two app processes. Figure 7.2 illustrates abstractly a transaction between *App A* (sender) and *App B* (receiver). To initiate the transaction, *A* writes its transaction data via `/dev/binder` to the Binder kernel module (step 1). The transaction data contains a token (`recv`) identifying the communication peer (i.e., *B*) as well as some payload containing, e.g., the method ID to be executed by the receiver plus the method arguments (e.g., an Intent object). The kernel module then resolves the token to identify the recipient of the transaction, i.e., *B*, and extends the transaction data with the sender UID, i.e., UID of *A*. Afterwards, the module copies the transaction data into the user space of an *IPC Thread* selected from the IPC thread pool of *B* (step 2). If the caller expected a reply (*two-way transaction*), the reply is sent back to the caller via the kernel module (steps 3 and 4). A two-way transaction is implemented as a *closed wait*, i.e., the sender thread blocks until it receives a response and the kernel module ensures that this response originates from the receiver thread. In the case of a *one-way transaction*, the sender immediately receives a dummy reply to prevent blocking (not depicted in Figure 7.2).

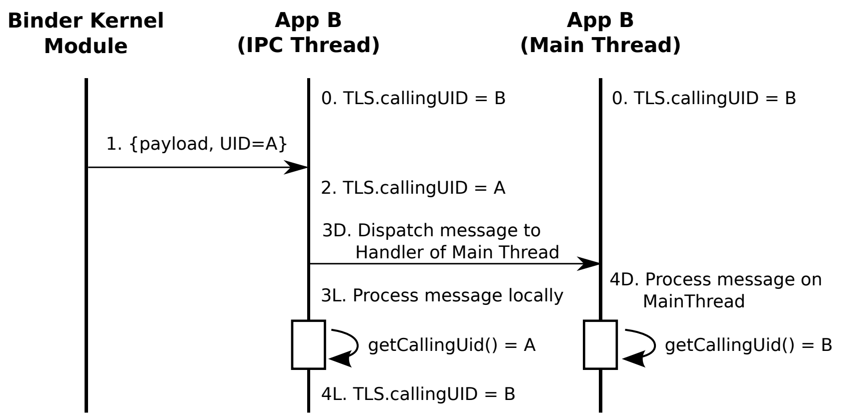

Figure 7.3: Handling of Binder transactions by the receiver and availability of caller UID.

Providing the sender UID to the receiving component is pivotal for enforcing permissions in Android's security design. First, system services and apps, which implement the application framework API, use this information to perform runtime checks (i.e., `PackageManager.checkPermission(permission, uid)`) whether calling apps hold the required permissions to access their interfaces. Using runtime checks enables these services to enforce permissions flexible and at the granularity of interface functions instead of app components. For instance, the *LocationManager* requires either the Permission `FINE_LOCATION` or `COARSE_LOCATION` depending on the parameters passed to the API call. Second, system services like the *ActivityManagerService* mediate between caller and callee applications (e.g., when an app queries a remote ContentProvider or when delivering an Intent) and these system services use the caller UID when mediating to check whether the caller is allowed to access the callee.

7.5 Technical Problem Description

As explained in Section 7.4.1.2, Binder provides IPC message recipients with the UID of their *direct* caller. However, as also shown in that Section, Android introduced different abstraction layers for Binder IPC to enable an *inter-component* communication on top of the inter-process communication. These abstractions introduce *indirections* and *message dispatches* that cause the caller UID provided by Binder to be lost along ICC control flows or to be insufficient.

7.5.1 Message Dispatching

To understand how the caller UID provided by Binder can be dropped in ICC, one first needs to examine how an incoming IPC transaction is handled at the receiver side. Figure 7.3 extends Figure 7.2 after the receiver (*App B*) has received the transaction (Step 2 in Figure 7.2 and Step 1 in Figure 7.3). The IPC thread of *App B* that is selected to handle this incoming transaction copies the sender UID of the transaction (i.e., *UID=A*) into its thread-local storage (TLS; step 2). Any application code executed on this thread can query this sender UID through the `Binder.getCallingUid` function. However, if the thread is *not* processing a Binder transaction such that the TLS gets never updated, this function call defaults to the threads own information (e.g., `Binder.getCallingUid` would return the UID of the thread itself, as set in step 0).

A received transaction can be processed in one of two possible ways and depends on the targeted component type: First, the message could be handled locally in the context of the IPC Thread (step 3L), which preserves the IPC provenance information. A *Service* or *ContentProvider* component, for instance, would be executed by default in this context. As a consequence, these components can call `Binder.getCallingUid` at any time and retrieve the process UID that triggered their current execution. As explained in Section 7.4.1.2, this is pivotal for enforcing the default permissions in Android's system services. Once the execution of app code on the IPC thread has finished, this thread resets its TLS (step 4L).

Alternatively, the IPC Thread can dispatch the processing of a received message to a worker thread (by default the application's *Main Thread*[1]). The message is dispatched by means of Android's *Handler* mechanism[2] (step 3D). For instance, this worker thread typically performs the handling of received Intents, which includes the processing of received Broadcast Intents, executing IntentServices, as well as *Activity*-related Intents (step 4D). However, the worker thread is executed in a different context and naturally with a different TLS. Thus, calling `Binder.getCallingUid` on the worker thread will always result in the retrieval of the worker thread's process UID. Dispatching the message will, therefore, effectively drop IPC provenance information: Components executed on the application's worker thread will have no information about which process has triggered their current execution and, hence, cannot distinguish whether their execution is legitimate and whether they can trust any received payload.

7.5.2 Indirect Communication

Android's model for ICC also introduced indirect communication between components, which renders Binder's IPC provenance information insufficient, in particular in the case of Intent-based ICC. As mentioned in Section 7.4.1.1 and illustrated in more detail in Figure 7.4, the *ActivityManagerService* is responsible for relaying Intents between apps. Thus, the actual communication between Intent senders and Intent receivers consists of two distinct Binder transactions. As a consequence, Binder's IPC provenance mechanism will at the receiver's side always identify the *ActivityManagerService* as the IPC caller, instead of the actual origin of the received Intent (i.e., the UID of the Intent sender). An exception from this shortcoming are *Activity*-related Intents that require a return value from the receiver. In that specific case, the Intent receiver can request the sender ID from the *ActivityManagerService*.

7.5.3 Provenance Information vs. Permissions

Many attacks [148, 44, 30] that have been reported in the Android security literature can be effectively mitigated if the callee is provided with comprehensive IPC provenance information. In rare cases [44], the cause for the discovered vulnerability was simply a

[1] Also referred to as *UI Thread* or *Activity Thread*.

[2] `http://developer.android.com/reference/android/os/Handler.html`

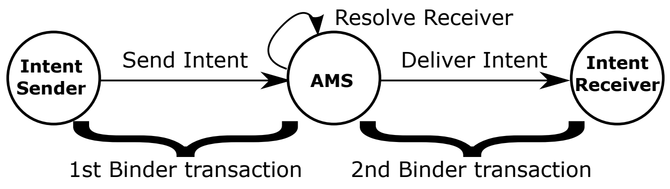

Figure 7.4: Indirection in Intent-based ICC.

forgotten permission check. In general, however, the situation is more complex. The confused deputy attacks presented in [148] relied on a privileged *BroadcastReceiver* that modified the system state (e.g., Wi-Fi or GPS state) on behalf of any broadcast sender. The receiver did not check whether the sender was entitled to send this command, since the current Android design does not provide the technical means that enable the *BroadcastReceiver* to retrieve the caller's UID and to endorse the caller for this privileged command. As a potential fix, a new permission could be introduced to protect this receiver from receiving Intents from unprivileged apps. However, this approach would be very inflexible: it would require a new permission for every distinct privileged receiver, especially when a receiver holds multiple privileges (e.g., Wi-Fi *and* GPS) or the exact privileged operation depends further on message parameters. We assume these to be the reasons why the vulnerabilities mentioned above are still not fixed in Android v4.2.2, although they have been known since v2.2 [148]. In contrast, provisioning comprehensive IPC provenance information as in SCIPPA provides the means for a flexible and fine-grained access control to (system) app developers and is, thus, preferable to effectively prevent attacks, such as confused deputy attacks [148, 30].

7.5.4 Broader Context of this Thesis

In addition to the above mentioned problems for Android's default security design, those shortcomings in Android's IPC provenance provisioning also affect all previously presented solutions in Chapters 3 through 6 of this thesis. Like the default permission check, the middleware enforcement points in those solutions ultimately rely on Android's Binder IPC to provide the identity of calling, remote applications—i.e., the subject's identity—for which they then retrieve the security context (e.g., the app's domain in TRUSTDROID or the subject type in FLASKDROID). The lack of the required provenance information in the above mentioned scenarios prevents those solutions from being applicable in those specific scenarios, for instance, enforcement points in *Activity* or *Broadcast Receiver* components are unable to identify the subject and hence unable to enforce any security policy. Thus, by addressing the shortcomings in Android's IPC provenance provisioning, not only the default security architecture is improved but also a more solid and reliable foundation for security extensions (such as the ones presented in this thesis) is laid.

7.6 Requirements Analysis

In this section, we define our adversary model, derive requirements and discuss challenges for a comprehensive Binder IPC call provenance on Android.

7.6.1 Adversary Model

The attacker model for our design of SCIPPA considers a strong attacker that is able to mount confused deputy, Intent hijacking and Intent spoofing attacks.

Confused Deputy Attacks. We adopt the confused deputy attacker model [88] that was adapted to the specific scenario of Android [36, 148]. In this model, a malicious app with an insufficient set of permissions for its malign purpose tricks a privileged app into executing its privileges on behalf of the malicious app. For instance, Enck *et al.* [44] reported the possibility of sending an Intent to the *Phone* app in order to start a phone call without holding the corresponding `CALL_PHONE` permission. Porter Felt *et al.* [148] discovered several *BroadcastReceiver* components in system services that acted as confused deputies and allowed any app to change, for example, the Wi-Fi or GPS status.

Intent Hijacking and Spoofing. In addition to confused deputy attacks, we consider Intent hijacking and Intent spoofing attacks [21]. To mount an Intent hijacking attack, the attacker registers an app in the system such that (ordered) broadcast Intents or certain *Activity*-related Intents are delivered to this registered app instead of to the actually intended recipient. Thus, the malicious app is able to receive any information contained in the Broadcast. Additionally, in the context of Activities, the attacker can use this technique to mount phishing attacks by spoofing legitimate Intent messages. Intent spoofing attacks are similar to the confused deputy attacks described above in which the recipient does not verify the origin of the received Intent and potentially takes inappropriate actions.

Restrictions. We focus in this work exclusively on *Binder IPC* and exclude other IPC channels such as files or sockets. A solution for these alternate channels would require additional extensions to low-level services such as the filesystem. Moreover, we limit our solution to *direct* IPC and do not consider covert channels as leveraged in collusion attacks [161, 117]. Finally, we do not explicitly consider attacks that compromise the system integrity such as root exploits or attacks against system components. However, we consider 3^{rd} party apps to be in full control of their sandbox. Apps can include native code that is able to rewrite the application code with techniques used, e.g., for inlining reference monitors [210, 7].

7.6.2 Requirements and Challenges

In this Section we derive the necessary requirements for SCIPPA and elaborate on technical and conceptual challenges in context of the Android system model.

Availability of Provenance Information. Due to message dispatching, provenance information is currently only available to app components that are executed in the context of a receiving IPC thread. Thus, the first requirement for a comprehensive solution is to extend Android's app model to propagate IPC provenance information to all app components. A particular challenge to be addressed in this context is to identify the correct IPC context of each thread. The Main Thread, for instance, usually handles workloads dispatched by multiple IPC threads. Hence, its current IPC context depends on the IPC thread it is currently serving.

Building System-Centric IPC Call-chains. By default, Binder provides apps with the UID of their direct caller. However, when considering the indirections in Android's Intent-based ICC, this information is insufficient for callees to identify the initiator of incoming requests. This leads to the second requirement: it is necessary to establish system-centric call-chains for Binder IPC so that we can provide receivers of Binder transactions with provenance information. This would enable receivers to answer questions like *"Who sent this Intent?"*.

Returning Call-Chains to Senders. While Android provides the receiver of a Binder transaction with limited means to retrieve the sender's UID, it does not provide any feedback to the sender on how their message was handled in the system. This missing feedback makes the senders unaware of, e.g., Intent hijacking and phishing attacks [21]. The third requirement is, therefore, to establish a feedback mechanism for IPC senders by returning already established, finalized call-chains to them. This enables the senders to analyze how their message was handled both by the system and other applications and, thus, to *detect* potential hijacking and phishing attacks. A technical challenge is to efficiently address branching of call-chains, which leads to a *1:N* communication (e.g., when broadcasting an Intent).

Tagging Asynchronous Messages. Although Binder transactions are synchronous, the protocols and mechanisms Android deploys on top of Binder can be asynchronous. For instance, *sticky* Broadcast Intents are kept in the system and are delivered even to recipients that register *after* the broadcast was sent. Thus, to effectively fulfill the first three requirements, this asynchronicity needs to be addressed, e.g., by tainting asynchronously delivered messages with their associated IPC provenance information.

7.7 System-Centric IPC Call-Chains

In this section, we describe our extensions to Android's Binder IPC at the kernel and user space to establish call-chains along *direct* ICC control flows. Further, we elaborate on extensions to Android's message handling mechanism to propagate those call-chains between the threads of an app.

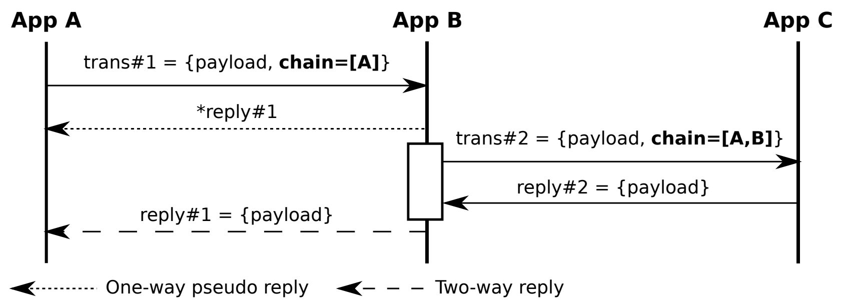

Figure 7.5: Call-chains during recursive Binder IPC calls.

7.7.1 Establishing Call-Chains

At the core of our solution are extensions to the Binder kernel module. In Binder, IPC messages are passed between processes as Binder transactions. Our extensions construct call-chains across app processes by linking the transactions along a direct thread of control for inter-application communication. Figure 7.5 illustrates recursive Binder transactions between three apps *A*, *B*, and *C*. For instance, *App A* could be an Intent sender, *App B* the *ActivityManagerService*, and *App C* the Intent receiver (cf. Figure 7.4). Moreover, Figure 7.5 shows that we differentiate in our design between *two-way* and *one-way* transactions. In recursive two-way transactions the second transaction `trans#2` is nested in the first transaction `trans#1`. In contrast, in one-way transactions `trans#1` is finished before `trans#2` is triggered, as illustrated by the pseudo reply `*reply#1`.

Binder transactions. The operations performed within the Binder kernel module in this scenario are depicted in Figure 7.6. In general, for each process that registers with Binder as a sender/receiver, the kernel module sets up `Binder Proc` information associated with the process as a whole and `Binder Thread` information associated with a particular thread of the application process (i.e., threads from the app's Binder IPC thread pool and the main thread). When *App A* sends a message to *App B*, the Binder kernel module creates new transaction data `Transaction#1`, where `transaction_data` contains the message, `from_parent` and `to_parent` point to parent transactions at the sender's and receiver's side in case of recursive transactions, and `from_thread` and `to_thread` point to the sender's and receiver's involved `Binder Threads`. The `transaction_stack` attribute of `Binder Thread` points to the last processed (i.e., last sent or received) transaction of the associated thread. When an IPC thread of *App B* is ready to receive this transaction, the message is copied from the kernel to the thread's user space. At this point the kernel module provides the receiver thread with the sender UID (cf. Step 4 in Figure 7.2 on page 144), which is retrieved from the kernel task information associated with the transaction sender's `Binder Proc`.

Constructing Call-Chains in Two-Way Transactions. As shown in Figure 7.5, *App B* issues a recursive call to *App C* while handling the call from *App A*. In Figure 7.6 this is depicted as `Transaction#2`. The attributes of `Transaction#2` are set the same

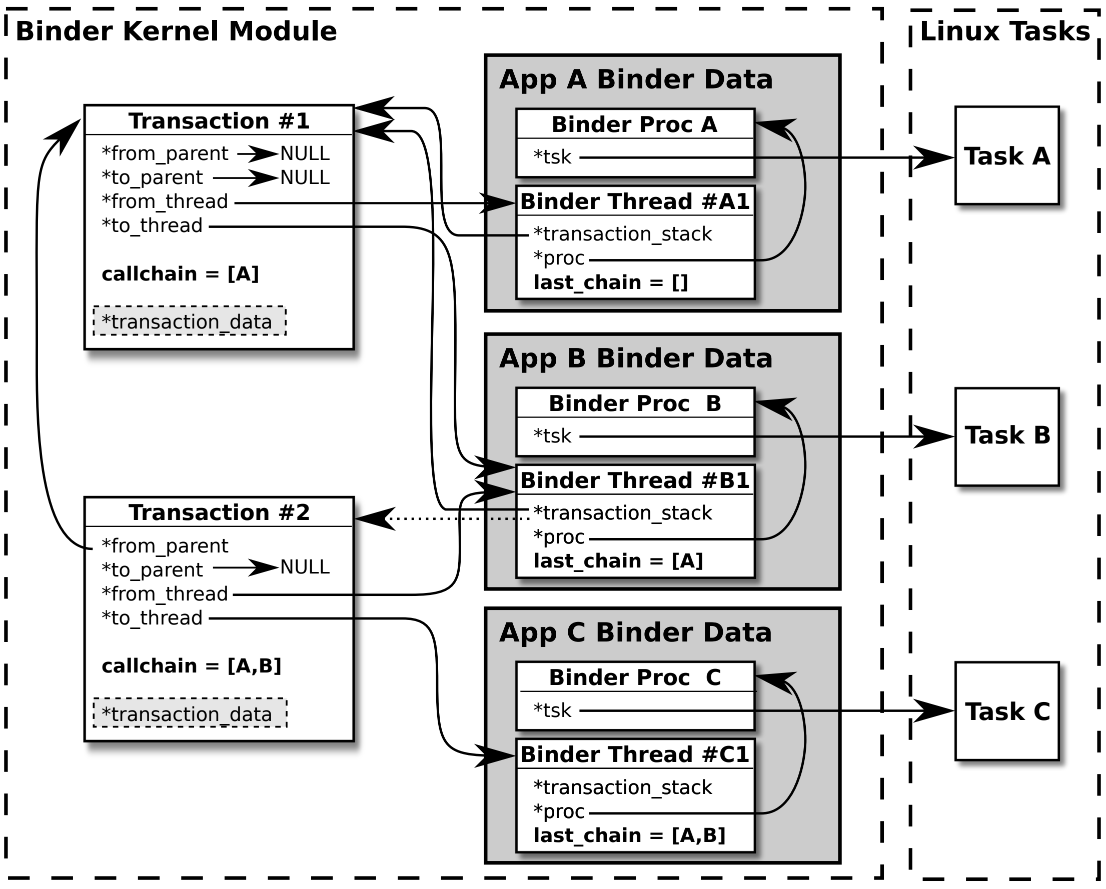

Figure 7.6: Constructing call-chains in recursive transactions. The dotted line represents the pointer *after* Transaction#2.

as those of `Transaction#1`. However, since `Transaction#2` is the most recent transaction for *App B*, its `transaction_stack` pointer is adjusted accordingly. The dotted line represents the pointer *after* `Transaction#2`.

In SCIPPA, we attach the current state of the call-chain to each transaction data. To this end, we extended the `transaction` data structure with new attributes to hold a call-chain. Every time a recursive call is made, the kernel module copies the call-chain from the direct predecessor transaction to the new transaction (or starts a new call-chain if no predecessor exists) and appends the UID of the current sender to the call-chain. In addition to the UID, we also save the sender's process and thread ID in the call-chain and assign every new call-chain a unique ID. This secondary information serves primarily for the analysis of chains constructed in SCIPPA.

Constructing Call-Chains in One-Way Transactions. For one-way transactions, the mechanism depicted in Figure 7.6 is not applicable, since `Transaction#1` is freed before `Transaction#2` is allocated. Thus, when continuing the chain, the data structure for the direct predecessor of `Transaction#2` no longer exists. To establish this missing link between transactions, we save the last received call-chain as part of the

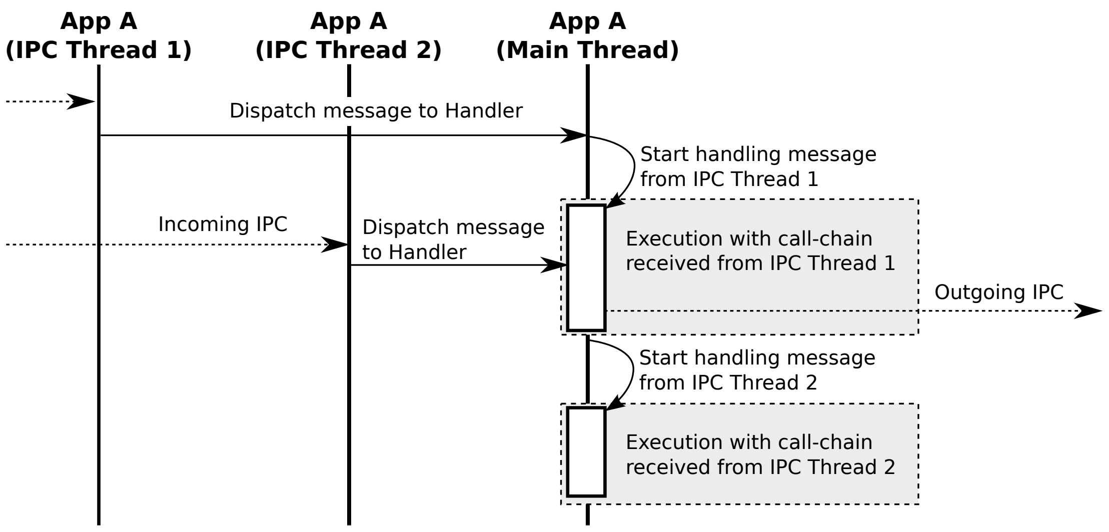
App A (IPC Thread 1)
App A (IPC Thread 2)
App A (Main Thread)
Dispatch message to Handler
Start handling message from IPC Thread 1
Incoming IPC
Dispatch message to Handler
Execution with call-chain received from IPC Thread 1
Outgoing IPC
Start handling message from IPC Thread 2
Execution with call-chain received from IPC Thread 2

Figure 7.7: Message processing by Handler on Main Thread.

`Binder Thread`. When the application code executing on the IPC thread sends the next transaction (i.e., transaction *B-C*), the saved call-chain is continued. This solution is based on the observation that IPC threads are at any given time either executing app code as direct consequence of the last received IPC call, or, when this execution has finished, return to a state in which they can receive the next call and hence next call-chain.

7.7.2 Intra-App Call-Chain Propagation

As explained in Section 7.5, certain types of IPC messages are dispatched by the receiving IPC thread to the Main Thread (or a dedicated worker thread) by adding it to the Main Thread's Handler message queue. The Main Thread handles queued messages sequentially (cf. Figure 7.7). Thus, the current IPC context of the Main Thread directly depends on the message currently processed. To propagate the call-chain received by the IPC thread to the Main Thread, we extended the `Message` and `Looper` classes of Android's application libraries to attach the received call-chain to new messages for the Handler. The Main Thread's `Handler` class is extended to always update its current IPC context with the call-chain of the message it processes next. Thus all outgoing IPC during this processing continue the call-chain correctly.

In Android's system services and app libraries exist several subclasses of the Handler class, e.g., a dedicated Handler in the *ActivityManagerService* or *PowerManager*. For our current implementation of SCIPPA, we focused on the most important handler for applications, the `ActivityThread` class, which is responsible for handling ICC actions such as processing received (broadcast) Intents, *Activity* life cycle events, or creating and binding to Service components.

7.7.3 Asynchronous Call-Chain Propagation

One particular challenge for establishing call-chains that include broadcast Intents are *sticky* broadcasts. As long as the sender app does not cancel the broadcast and does not get uninstalled, sticky broadcasts are stored by the *ActivityManagerService*. These broadcasts are even delivered to relevant Broadcast Receivers that register in the system *after* the broadcast Intent has been sent. To address this asynchrony within the control flow from the Intent sender to the receiver, we modified the *ActivityManagerService* to tag stored sticky broadcasts with the call-chain at the time the broadcast Intent was stored. Additionally, we modified the *ActivityManagerService*'s logic for delivering sticky broadcasts to adjust its current IPC context according to the call-chain stored with the sticky broadcast before sending and to restore its original IPC context afterwards. As a result, sticky broadcasts continue the call-chain so that its receivers can now identify the original broadcast sender.

7.7.4 Accessing Call-Chains from User Space

To provide the current call-chain information to the user space, we pass this information as part of the `binder_transaction_data` from the kernel to the IPC thread that receives the transaction. We extended the Android runtime library (cf. Section 7.4.1.1) to extract the call-chain from the transaction data and to subsequently store it in the thread local storage (TLS). From there it can be retrieved by any application code that is executed on the same thread. Similar to the default `getCallingUid` function, we introduce a new API function `getCallChainUids` to retrieve the call-chain. App developers can then retrieve information about those chained UIDs from the system—including the UIDs' package names, developer signatures, or permissions—and implement a fine-grained access control based on that information.

As explained earlier, some scenarios require that the user space is able to set its current call-chain. Therefore, we extended the `Binder` classes with functions to set the call-chain of the current thread. When a new Binder transaction is triggered, the set call-chain is passed to the Binder kernel module as part of the send transaction data. The kernel module uses this information to continue the call-chain unless it finds a call-chain derived from a direct two-way or one-way transaction, since they are prioritized over information received from the user space. To prevent the user space from forging or modifying call-chains, a token-based approach—i.e., user space processes only hold a *read-only* reference, bound to their UID, to retrieve the call-chain from the kernel space—could enable the kernel to verify that call-chains retrieved from user space were originally created by the kernel and hence to discard illicit chains.

7.7.5 Returning Call-Chains to Message Senders

Our extension to the kernel module propagates finished branches of call-chains back to the initial sender. We extended the Binder protocol with a new flag `BR_CALLCHAIN` to send a finalized branch of a call-chain back to the app that started this chain. To distinguish branches of different call-chains, the kernel module additionally provides the call-chain ID to the user space. While this mechanism returns all finalized branches to the sender, ongoing work extends Android's application model to efficiently store and manage this information. We are in the process of implementing a new application component type dedicated to this task. To transfer the call-chain information from an IPC thread to this new component, we plan an extension of the message dispatching mechanism.

7.8 Evaluation

In this section we evaluate and discuss the implementation of SCIPPA in terms of effectiveness and performance impact.

7.8.1 Experimental Methodology

All our experiments were performed on a standard Galaxy Nexus development phone. This platform is equipped with a TI OMAP 4660 chipset with an ARM Dual-core 1.2GHz Cortex-A9 CPU and 1GB of RAM. We implemented SCIPPA as a modification to the Android OS code base of v4.2.2_r1.2b and the Android Linux kernel in branch *android-omap-tuna-3.0-jb-mr1.1* with Binder protocol v7. Since the resolution of the Linux time facilities were too coarse grained for our microbenchmarks of Binder transactions, we leveraged the ARM *Performance Monitoring Unit* to calculate the performance overhead in CPU cycles. Since the PMU counts cycles per CPU core on our dual-core platform, we modified the Binder kernel module to acquire a spinlock for the duration of each measurement, thus eliminating the possibility that the current Binder thread is re-scheduled on the other CPU core while executing the measured code segment. Additionally, to be able to estimate the performance overhead in seconds, we adjusted the CPU frequency governor to always clock the CPUs at the maximum rate of 1.2GHz. Finally, to reduce the level of white noise in our measurements, we did not run any other apps except for our benchmark apps. Thus, all measurements approximate the *lower* bound for the actual overhead.

7.8.2 Performance Impact

To evaluate the performance impact of SCIPPA, we performed i) microbenchmarks of transactions in the Binder kernel module and ii) reimplemented relevant parts of the user space benchmarks of the closely related work *Quire* [36].

Transaction microbenchmarks. We performed microbenchmarks within the Binder kernel module for building and continuing call chains as described in Section 7.7. The results of our benchmarks are based on the measurements of 52,777 Binder transactions. Figure 7.8 presents the relative overhead vs. the data payload of the transaction. The maximum overhead was 2.23% and this overhead decreased with increasing payload size, where the memory copy operations for the data buffer outweigh the call-chain operations. However, when taking the frequencies of different payload sizes into consideration, the weighted average remains at 2.23%.

This small performance overhead is further illustrated in Figure 7.9, which shows the cumulative frequency distribution of the CPU cycles required for performing Binder

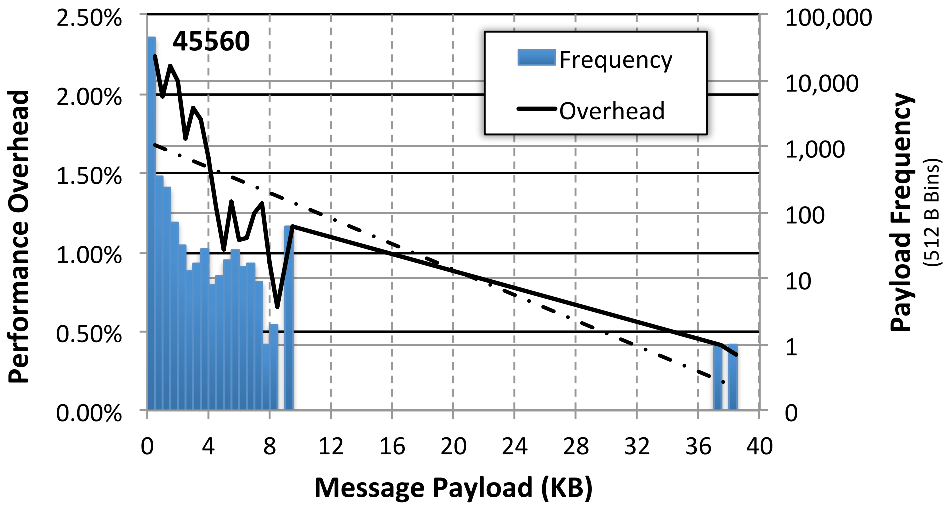

Figure 7.8: Performance overhead of Binder transactions vs. payload size and frequency breakdown of payload sizes.

transactions in SCIPPA and stock Android. On average, transactions on stock Android required 18,850 cycles and 99% of the measured transactions required less than 115,000 cycles. On SCIPPA this performance merely decreases to an average of 22,581.15 cycles per transaction, which translates to 18.82μs in our experimental setup, and 99% of the transactions required less than 130,000 cycles.

It should be noted that 0.58% of the measurements in our data set were extreme outliers that were in the range of 3 to 5 magnitudes higher than the rest of the measurements. These outliers occurred in both stock Android and SCIPPA benchmarks. We could trace these outliers back to thread blocking during parsing of the binder transaction header, i.e., *operations independent from our* SCIPPA *modifications*. However, since these rare outliers significantly distorted the mean and the margin of error in our measurements, we excluded them from the results presented here.

User space benchmark. Since applications in SCIPPA have to retrieve, parse, and set the received call-chain as part of their Binder IPC thread operations (cf. Section 7.7), we measured the overhead of SCIPPA from the application layer perspective. To this end and to provide a better comparison with existing work, we re-implemented the test cases presented in the closest related work *Quire* [36]. In this test, the Service components of several test apps interact a) to pass a message with variable size payload roundtrip between two apps and b) to send a message without payload roundtrip between multiple apps to build call-chains of different lengths.

Figure 7.10 presents the average performance per roundtrip versus the message payload as computed from 11,000 measurements per payload size. The payload size ranges from 0 bytes to 6,336 bytes in 64 bytes increments. In our data set, SCIPPA imposed between 3.70–25.33% overhead, which is comparable to Quire's performance (21% slowdown). Figure 7.11 shows the average performance per roundtrip vs. the

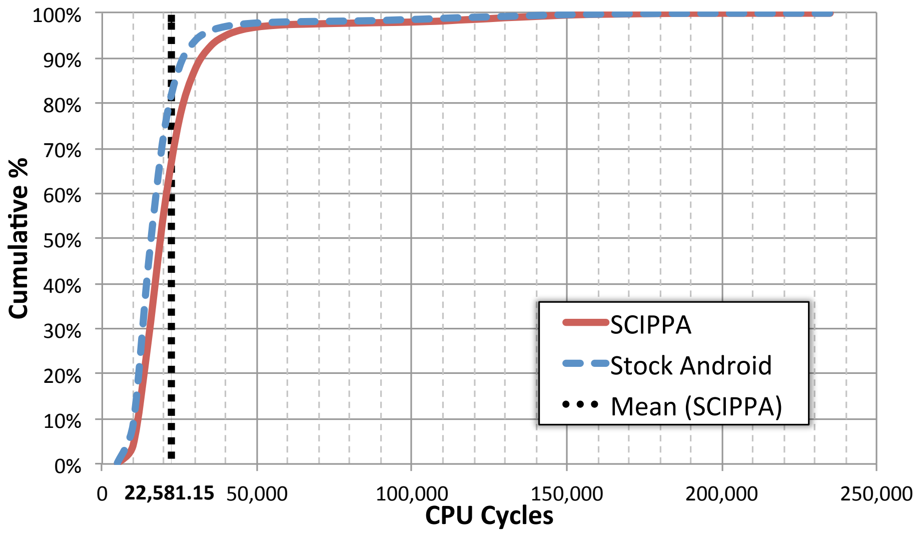

Figure 7.9: CDF of CPU cycles for Binder transactions on SCIPPA and stock Android. Vertical dashed line indicates the mean execution time for SCIPPA.

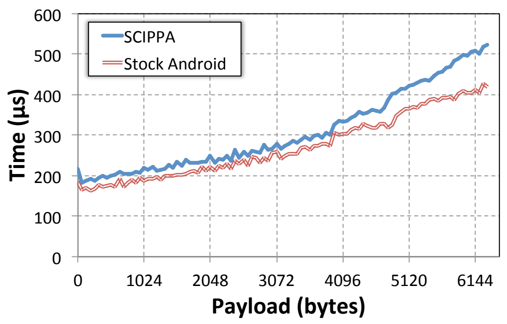

Figure 7.10: Single Service call roundtrip time vs. payload size.

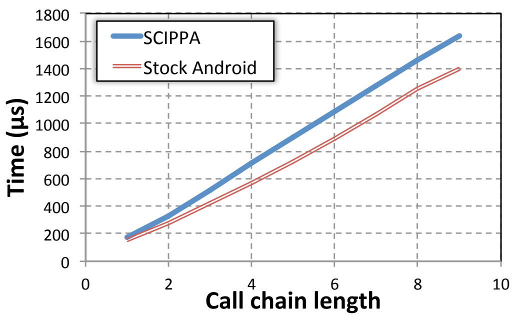

Figure 7.11: Service call roundtrip time vs. call chain length.

call-chain length with a max length of 9 (i.e., 10 apps involved) and 11,000 measurements per length. SCIPPA's overhead is 12.70–26.73%, which is again comparable to Quire (20–25% slowdown).

7.8.3 Binder IPC Provenance

In this section, we provide statistics on the call-chains observed in SCIPPA during our tests and evaluate how well these call-chains provide the necessary IPC provenance information to efficiently mitigate the different attacks introduced in our requirements analysis (cf. Section 7.6).

Call-chain Statistics. Table 7.1 summarizes statistics on call-chains observed during our testing. All margins of error are for a 95% confidence. We logged in total 54,670 call-chains with an average length of 1.56. All chains had at least two branches with

General:	
#Call-Chains	5,4670
Chain length	1.56 ± 0.01
Max. length	13
Branching:	
#Chains with branches	54,760 (100%)
#Branches (total)	141,330
#Branches (per chain)	2.59 ± 0.08
Max. number of branches	1,194
Dispatching:	
#Chains with dispatching	3,237 (5.91%)
#Dispatches (total)	24,966
#Dispatches (per chain)	7.71 ± 1.92
Max. number of dispatches	2,784

Table 7.1: Call-chain statistics.

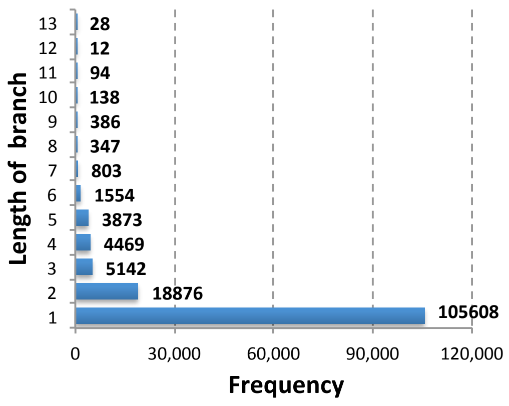

Figure 7.12: Break down of observed call-chain lengths.

2.59 being the average number of branches per chain. Figure 7.12 provides a breakdown of the branch lengths and shows that the maximum observed length is 13 and that chains with a length of one are most frequent. The max number of branches for one chain is 1,194. Additionally, 5.91% of all chains contained at least one dispatch between an IPC thread and the Main Thread. On average, each chain contained 7.71 dispatches with 2,784 being the highest number of dispatches observed for one chain.

Attack mitigation. To verify that SCIPPA fulfills the requirements stated in Section 7.6.2, we developed a set of interacting test applications, which implement different combinations of inter-component communication that model the scenarios that have been reported to be prone to attacks such as confused deputy attacks [148] or Intent hijacking [21]. This includes common inter-app communications such as starting Activities, Broadcast Intents, or binding and calling Services including IntentServices. In general, all called components were able to retrieve the call-chain of the direct thread of control that lead to their invocation and thus identify the initiator of the call-chain. Also, all senders were successfully notified by the kernel module about finalized call-chain branches. Based on this information, we were able to implement a per-UID access control that is more fine-grained and flexible than Android's static Permission system.

Since the most well-known reported confused deputy attacks rely on Broadcast Receivers [148], we briefly elaborate on call-chains for broadcast receivers in our testbed. Figure 7.13 shows an established call-chain for a single Broadcast Intent sent by the app with UID 10043 (lower left corner). The Intent is sent to the *ActivityManagerService* as part of the system server with UID 1000, where the task to send this broadcast is dispatched to a dedicated thread (1000:403:777 → 1000:403:420). This thread delivers the Broadcast Intent in parallel to all *dynamically* registered receivers (upper left rectangle) and in *order*[3] to all receivers registered statically through the applications' manifests (lower right rectangle). Each app receives the Intent via an IPC thread and processing of the Intents by the Broadcast Receiver components is dispatched to the apps' Main Threads.

As a consequence, each Broadcast Receiver is able to retrieve the branch of the call-chain that lead to its invocation and, hence, to identify the sender of the broadcast. For instance, the receiver of UID 10047 retrieves the call-chain 10043 → 1000 and the receiver of UID 10045 retrieves the chain 10043 → 1000 → 10044 → 1000 that shows all receivers previous to itself in the ordered delivery. Using this information, Broadcast Receivers in Scippa can now efficiently evaluate their trust in received messages and their senders, which allows them to react accordingly by, e.g., refusing to accept spurious messages. In addition to our test cases, we also verified that the privileged Broadcast Receiver that was reported as a confused deputy in [148] is now able to identify the broadcast Intent sender and hence to apply fine-grained access control depending on the Intent payload (e.g., GPS vs. Wi-Fi control commands). That eliminates the confused deputy vulnerability without the need to split its component interface or to introduce new permissions.

Additionally, the sender received from the kernel module four `BR_CALLCHAIN` notifications about the call-chain branches that ended with the apps with UIDs 10045 through 10048. Thus, the sender is able to identify the receivers of its broadcast and to determine if its broadcast was potentially hijacked by an unintended receiver [21]. In case of ordered broadcasts, it can even determine which app was responsible for cancelling the further delivery of the broadcast.

7.8.4 Discussion and Limitations

The most important limitation for the effectiveness of our approach is that the call-chain can be lost if communication between threads occurs over channels currently not covered by Scippa. With message dispatching, Scippa covers one of the major communication channels between Android application threads, but other channels exist (e.g., `notify`). Future work has to address these channels through extensions to the Dalvik VM and Java language classes in the Android framework. For instance, initial experiments have shown that it is possible to forward the call-chain to newly spawned threads, e.g., for *IntentService* components or *AsyncTasks*.

[3]Broadcast receivers registered through the manifest are *always* served in ordered fashion, but intermediary receivers only can stop further delivery when the *ordered* flag is set.

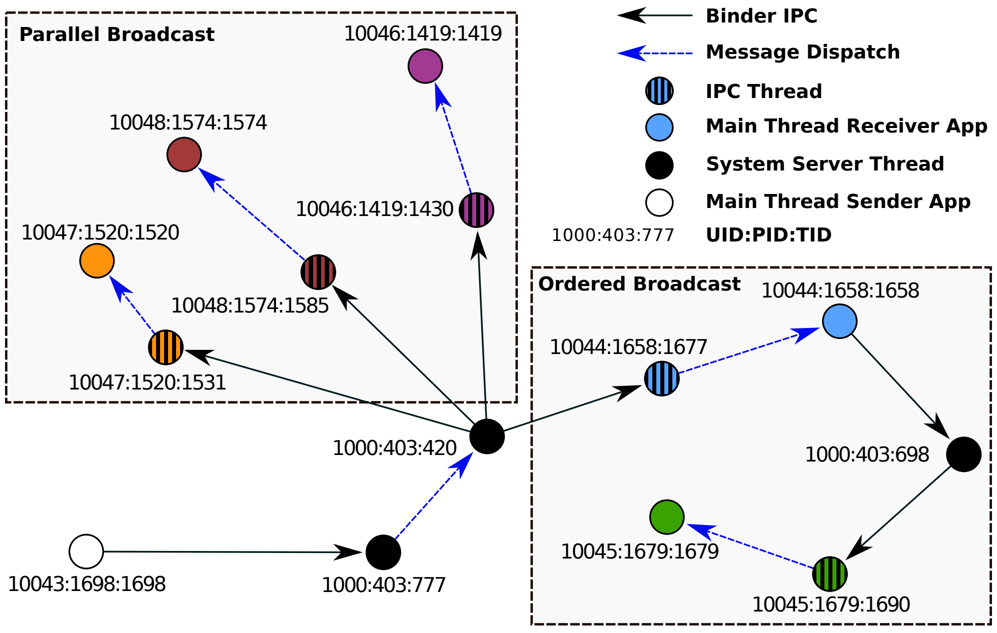

Figure 7.13: Call-chain for parallel and ordered broadcasts.

Similarly, SCIPPA currently only covers direct control flows for ICC. Hence, indirect control flows are an open problem. Providing an efficient solution to address indirect control flows is an orthogonal problem and affects other approaches such as dynamic taint-tracking [46] as well.

Since our solution relies on code within the app sandboxes to forward call-chains on message dispatching, this code base is prone to attacks by malicious apps. While forging and modifying call-chains can be prevented with a token-based approach (cf. Section 7.7.4), the deliberate dropping of call-chains cannot be prevented. However, because UIDs are attached by the kernel to the call-chains during the sending of IPC calls, a malicious app can only hide previous hops in the chain but not itself. Thus, a malicious app cannot fool a receiver into trusting it by hiding its predecessors in the call-chain [36] and this is primarily a problem if multiple malicious apps collude [117, 161, P5].

7.9 Related Work

IPC-based domain isolation. Thread-migrating IPC has been used in high-assurance systems [107, 169, 163, 98] as building block for domain-based isolation by factoring applications into smaller domains. Domains are usually compartmentalized at process boundaries and IPC is used to connect them. The security of IPC has been investigated from different angles, e.g., for synchronous IPC [168] or language-based security [53]. Android borrows ideas from this literature (e.g., UID-based app sandboxes, connected via thread-migrating IPC). However, in this work we identified and addressed misaligned security assumptions when *Inter-Component Communication* is built on top of IPC.

Provenance frameworks. Establishing provenance information has been primarily investigated for data provenance [171, 121] and in distributed systems [204]. Specific to smartphones, the *SPADE* framework [64] has been ported to Android [95]. It uses the Binder debug interfaces to profile the IPC on Android and generate useful traces for device auditing. Scippa provides very similar information, however, in contrast to SPADE on Android, Scippa also provides the links between IPC channels along direct ICC control flows and thus valuable information for a more detailed auditing.

A recent approach called *EPIC* [137] uses static analysis to detect all potential Intent-based communication channels between application components. The information created by Scippa includes this information as well. However, Scippa only reflects actual runtime behavior. A combination of these two approaches could lead to a more comprehensive app testing, in which static analysis shows all potential channels and Scippa fills the gaps in this analysis (e.g., when a target cannot be resolved statically).

Android security. Research has established a large body of literature on Android security. With respect to preventing confused deputy attacks, related work [148] has proposed the poli-instantiation of apps in ICC to reduce the callee's privileges to the ones of their caller. XManDroid [P5] monitors all IPC communication and applies at runtime Chinese Wall security policies to prevent communication that could lead to a dangerous information flow. While poli-instantiation and XManDroid rely on strictly restricting privileges or communication channels, Scippa and closely related work (*Quire* [36]) rely on provisioning IPC provenance information to callees to enable them to apply fine-grained access control on their ICC interfaces. This allows them to securely provide APIs to other apps. In contrast to Scippa, however, Quire's prototypical implementation requires developers to explicitly extend all Stub and Proxy interfaces in order to construct call-chains. Scippa abstains from a developer-centric approach for establishing IPC provenance information and implements a system-centric solution that builds call-chains transparently to developers. Moreover, in Quire's developer-centric approach, call-chains are created and extended within the app code and, hence, this approach requires verifiable statements to establish trust in and authenticity of chains. In Scippa, the kernel creates and extends the call-chains and only when chains are propagated through the user space back to the kernel, a lightweight cryptographic mechanism (e.g., tokens) is required to ensure authenticity of chains. Additionally, as a kernel-based solution, Scippa covers even cases in which apps do not use Stubs, but instead app code (e.g., native libs) communicates directly with the Binder module.

Besides confused deputy attacks, related work has proposed a solution [102] to mitigate Intent hijacking by applying heuristics-based access control for Intents to prevent their unintended delivery. While this is a system-centric, preventive security extension, Scippa provides a different trade-off. Scippa adds measures that allow an application to detect (not prevent) such attacks, but in turn provides a higher precision than heuristics due to its call-chain information.

Information flow control. Concepts from decentralized information flow control [129, 130, 129], e.g., as implemented in the DEFCON [123] and Asbestos [42] operating systems, have been applied within different solutions on Android. Most prominent solutions based on dynamic taint analysis include *TaintDroid* [46], *AppFence* [93], and *Paranoid Android* [149]. In contrast, Scippa does not aim at restricting information flows of sensitive data at information sinks, but instead aims at providing apps with IPC provenance information that enables them to effectively apply access control for sensitive data and functionality.

7.10 Conclusion

In this work, we presented Scippa, our architecture for provisioning Binder IPC provenance information on Android. It allows app components to identify the sending app of incoming IPC messages despite indirections and message dispatching. Using this provenance information, apps are now able to effectively apply per-sender access control to their interfaces. In contrast to related work, Scippa constitutes a system-centric approach that directly addresses conceptual shortcomings in Android's multi-layered inter-application communication. We presented an implementation of Scippa based for Android v4.2.2 and the evaluation of our prototype showed that Scippa imposes only minimal overhead when compared to stock Android. In addition, we deem the lessons learned from Scippa valuable for the design of future multi-layered OS security architectures that rely on thread-migration and that support liberal inter-app communication. In addition to its security benefits, Scippa also produces information that we deem valuable for areas of independent interest such as system analysis or forensics.

8 Conclusion

In this dissertation, we presented a line of work that adds mandatory access control (MAC) to the Android OS and in particular to its extensive middleware layer. Android's prior lack of mandatory access control has left room for successful attacks against the system integrity and also the user's privacy. Moreover, this lack disqualified Android for deployment in higher-security contexts, such as the increasingly popular dual-usage smartphones for enterprise deployment or the emerging market for governmental mobile devices. Focus of this dissertation was to explore the security design space of historical, established operating systems (e.g., the development of the Linux Security Modules) and transfer this knowledge to the particular design of the Android OS in order to retrofit Android's design with mandatory access control for enforcing improved security policies that protect the system integrity and the user's private information as well as enable advanced security models on Android. Further, a particular aspect of this work was to explore to which extent the strongly high-level API-oriented design of modern, mobile operating systems provides a better opportunity to more efficiently establish a higher security and privacy standard than is possible on current commodity PC platforms. A particular technical challenge of this line of work was to consider and consolidate enforcement of security policies at both the operating system level (i.e., Linux kernel MAC) and middleware level (i.e., our security extensions to Android).

Our work on establishing mandatory access control on Android stretched across the multiple peer-reviewed publications [P4, P5, P3, P2, P1], which each contributed to the development of our MAC for Android as presented in this dissertation: Our TrustDroid architecture (see Chapter 3 on page 15) is one of the first security extensions to Android that specifically targeted the multi-personna use-case where devices are used in both private and business contexts. It introduced a novel mandatory access control architecture for Android's software stack that extended core middleware services and the Linux kernel to enforce static security policies that isolate private data and apps from business-related data and apps. We extended TrustDroid's architecture in our XManDroid work (see Chapter 4 on page 39) to mitigate the threat of application-level privilege escalation attacks, i.e., collusion or confused deputy attacks, using a novel policy language based on the VALID policy language for infrastructure cloud environments. Our FlaskDroid solution (see Chapter 5 on page 59) is the first generic security architecture for the Android OS that can serve as a flexible and effective ecosystem to instantiate different security solutions for the Android software stack. Its design is inspired by the Flask security architecture, where various Object Managers at middleware and kernel-level are responsible for assigning security contexts to their objects and enforce access control on them. A key observation of the FlaskDroid work is, that almost all proposals for Android security extensions in the existing literature (including our TrustDroid and XManDroid) constitute mandatory access control mechanisms that are tailored to the specific semantics of their addressed security problem. By introducing a generic security architecture on top of SE Android together with an efficient policy language (inspired by SELinux) that takes the specifics of Android's middleware semantics into account, FlaskDroid enables a policy-driven instantiation of existing and new security/privacy models. Although FlaskDroid was the first generic security architecture for Android and was able to instantiate different security models, its design still confined policy authors to the semantics of the FlaskDroid policy language. To remedy this situation, we introduced our Android Security Framework (ASF; see Chapter 6 on page 109), a generic, extensible security framework for Android that enables the development and integration of a wide spectrum of security models in form of code-based security modules. The design of ASF reflects lessons learned from the literature on established security frameworks on commodity platforms (such as Linux Security Modules or the BSD MAC Framework) and intertwines them with the particular requirements and challenges from the design of Android's software stack to build a policy-agnostic and multi-tiered security infrastructure. ASF provides a novel security API that supports authors of Android security extensions in developing their modules and supports implementation of policy logic that resembles mandatory result automata. Lastly, our Scippa extension to the Android IPC mechanism (see Chapter 7 on page 139) establishes inter-process communication (IPC) call-chains across application processes. Thus, although Scippa is itself not an access control mechanism, it contributes greatly to the domain of access control on Android by providing essential information: IPC provenance information required to effectively prevent recent attacks. Any kind of enforcement point within the Android middleware relies on such provenance information to identify the subjects in access control decisions.

Future research directions. The author of this dissertation envisions different future research directions that continue this line of work. First, all contemporary security extensions to Android's middleware, and specifically its application framework, have been integrated in a best effort approach, but no security guarantees can currently be given. Recent advances in static analysis of Java code should be applied to analyze the Android application framework's security- and privacy-relevant data and control flows. Eventually, this should allow a security analysis of the Android middleware that gives security guarantees for existing and future security extensions (e.g., completeness and correctness of the authorization hook placement or optimization opportunities). Additionally, through establishing of the static execution context along those flows (e.g., constraints), this can also lead to valuable information that can guide future implementations of security extensions and allow by design better security guarantees. For instance, placement of authorization hooks in the Android middleware should be automated in the spirit of existing solutions for commodity systems [127, 61, 60].

Another envisioned direction of future research is the combination of our mandatory access control systems with capability systems in the spirit of, e.g., *Capsicum* [199] or *Cheri* [200]. Whether this approach is feasible (and beneficial) requires evaluation on how Android's design leans towards capability-based access control. However, on first glance the author is confident that Android's design is in accordance with the motivation behind (at least) the Capsicum architecture. For instance, Zygote already forms a process that manages capabilities for its application child processes; although on Android exists a stronger IPC-based interaction between the app and the system apps/services than in usual capability models. The current vision for this line of work foresees retrofitting Android's code base to work with data object references (or "capability tokens"). Another promising approach seems the transition from an access control list based permission system to capability-based permission system, including technical questions such as how capabilities can be efficiently transferred via Binder.

A general challenge for mandatory access control solutions is the generation of effective and "good" policies. Current approaches, such as EASEAndroid [194], demonstrated how the default approach to generating policies from audit logs can be improved and scaled to the demands of real-life deployments. Other solutions infer the required access control rules from the programmer expectations of code that executes on the platform [189]. However, the author considers this as an open problem and very valuable future research direction for improving access control systems such as the ones presented in this doctoral thesis.

Lastly, new deployment options for security extensions at application layer only should be explored by building on top of the recently developed app virtualization solution *Boxify* [S1]. In particular, the integration of security architectures like ASF or FlaskDroid into the Boxify virtualization layer (i.e., the virtual application framework services of Boxify) seems like a canonical solution. First envisioned application domain of this new approach is considered to be similar to TrustDroid by establishing a fortified application sandbox for private apps that are installed on a business device.

Bibliography

Author's Papers for this Thesis

[P1] Backes, M., Bugiel, S., and Gerling, S. Scippa: System-Centric IPC Provenance on Android. In: *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC 2014)*. ACM, 2014.

[P2] Backes, M., Bugiel, S., Gerling, S., and Styp-Rekowsky, P. von. Android Security Framework: Extensible Multi-Layered Access Control on Android. In: *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC 2014)*. ACM, 2014.

[P3] Bugiel, S., Heuser, S., and Sadeghi, A.-R. Flexible and Fine-Grained Mandatory Access Control on Android for Diverse Security and Privacy Policies. In: *Proceedings of the 22nd Usenix Security Symposium (SEC 2013)*. USENIX Association, 2013.

[P4] Bugiel, S., Davi, L., Dmitrienko, A., Heuser, S., Sadeghi, A.-R., and Shastry, B. Practical and Lightweight Domain Isolation on Android. In: *Proceedings of the 1st ACM CCS Workshop on Security and Privacy in Mobile Devices (SPSM 2011)*. ACM, 2011.

[P5] Bugiel, S., Davi, L., Dmitrienko, A., Fischer, T., Sadeghi, A.-R., and Shastry, B. Towards Taming Privilege-Escalation Attacks on Android. In: *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS 2012)*. The Internet Society, 2012.

Other Papers of the Author

[S1] Backes, M., Bugiel, S., Hammer, C., Schranz, O., and Styp-Rekowsky, P. von. Boxify: Full-fledged App Sandboxing for Stock Android. In: *Proceedings of the 24th USENIX Security Symposium (SEC 2015)*. USENIX Association, 2015.

[S2] Bleikertz, S., Bugiel, S., Ideler, H., Nürnberger, S., and Sadeghi, A.-R. Client-controlled Cryptography-as-a-Service in the Cloud. In: *Proceedings of the 11th International Conference on Applied Cryptography and Network Security (ACNS 2013)*. Springer-Verlag, 2013.

[S3] Bugiel, S., Davi, L., and Schulz, S. Scalable Trust Establishment with Software Reputation. In: *Proceedings of the 6th Annual Workshop on Scalable Trusted Computing (STC 2011)*. ACM, 2011.

[S4] Bugiel, S. and Ekberg, J.-E. Implementing an Application-Specific Credential Platform Using Late-Launched Mobile Trusted Module. In: *Proceedings of the 5th Annual Workshop on Scalable Trusted Computing (STC 2010)*. ACM, 2010.

[S5] Bugiel, S., Dmitrienko, A., Kostiainen, K., Sadeghi, A.-R., and Winandy, M. TruWalletM: Secure Web Authentication on Mobile Platforms. In: *Proceedings of the 2nd Conference on Trusted Systems (INTRUST 2010)*. Springer-Verlag, 2010.

[S6] Bugiel, S., Pöppelmann, T., Nürnberger, S., Sadeghi, A.-R., and Schneider, T. AmazonIA: When Elasticity Snaps Back. In: *Proceedings of the 18th ACM Conference on Computer and Communication Security (CCS 2011)*. ACM, 2011.

[S7] Bugiel, S., Nürnberger, S., Sadeghi, A.-R., and Schneider, T. Twin Clouds: Secure Cloud Computing with Low Latency. In: *Proceedings of the Communications and Multimedia Security Conference (CMS 2011)*. Springer-Verlag, 2011.

[S8] Ekberg, J.-E. and Bugiel, S. Trust in a Small Package: Minimized MRTM Software Implementation for Mobile Secure Environments. In: *Proceedings of the 4th Annual Workshop on Scalable Trusted Computing (STC 2009)*. ACM, 2009.

Technical Reports of the Author

[T1] Backes, M., Bugiel, S., Gerling, S., and Styp-Rekowsky, P. von. *Android Security Framework: Enabling Generic and Extensible Access Control on Android*. Tech. rep. A/01/2014. Saarland University, Apr. 2014.

[T2] Bugiel, S., Heuser, S., and Sadeghi, A.-R. *myTunes: Semantically Linked and User-Centric Fine-Grained Privacy Control on Android*. Tech. rep. TUD-CS-2012-0226. Center for Advanced Security Research Darmstadt, Nov. 2012.

[T3] Bugiel, S., Heuser, S., and Sadeghi, A.-R. *Towards a Framework for Android Security Modules: Extending SE Android Type Enforcement to Android Middleware*. Tech. rep. TUD-CS-2012-0231. Center for Advanced Security Research Darmstadt, Dec. 2012.

[T4] Bugiel, S., Davi, L., Dmitrienko, A., Fischer, T., and Sadeghi, A.-R. *XManDroid: A New Android Evolution to Mitigate Privilege Escalation Attacks*. Tech. rep. TR-2011-04. Technische Universität Darmstadt, 2011.

Other references

[1] Abrams, M. D., Eggers, K. W., LaPadula, L. J., and Olson, I. M. A generalized framework for Access Control: An informal description. In: *Proceedings of the 13th NIST-NCSC National Computer Security Conference*. NIST, 1990.

[2] Anderson, J. P. *Computer Security Technology Planning Study, Volume II*. Tech. rep. ESD-TR-73-51. Deputy for Command and Management Systems, HQ Electronics Systems Division (AFSC), L. G. Hanscom Field, Oct. 1972.

[3] *Android Malware Genome Project.* Online: http://www.malgenomeproject.org/.

[4] Andrus, J., Dall, C., Hof, A. V., Laadan, O., and Nieh, J. Cells: A Virtual Mobile Smartphone Architecture. In: *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP 2011).* ACM, 2011.

[5] ARM. *TrustZone.* Online: www.arm.com/trustzone.

[6] Azab, A. M., Ning, P., Shah, J., Chen, Q., Bhutkar, R., Ganesh, G., Ma, J., and Shen, W. Hypervision Across Worlds: Real-time Kernel Protection from the ARM TrustZone Secure World. In: *Proceedings of the 21st ACM Conference on Computer and Communication Security (CCS 2014).* ACM, 2014.

[7] Backes, M., Gerling, S., Hammer, C., Maffei, M., and Styp-Rekowsky, P. von. 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).* Springer-Verlag, 2013.

[8] Badger, L., Sterne, D. F., Sherman, D. L., Walker, K. M., and Haghighat, S. A. Practical Domain and Type Enforcement for UNIX. In: *Proceedings of the 16th IEEE Symposium on Security and Privacy (Oakland 1995).* IEEE, 1995.

[9] Badger, L., Sterne, D. F., Sherman, D. L., Walker, K. M., and Haghighat, S. A. A domain and type enforcement UNIX prototype. In: *Proceedings of the 4th Usenix Security Symposium (SEC 1995).* USENIX Association, 1995.

[10] Baker, D. B. Fortresses built upon sand. In: *Proceedings of the 1996 Workshop on New Security Paradigms (NPSW 1996).* ACM, 1996.

[11] Barr, K., Bungale, P., Deasy, S., Gyuris, V., Hung, P., Newell, C., Tuch, H., and Zoppis, B. The VMware Mobile Virtualization Platform: Is That a Hypervisor in Your Pocket? *SIGOPS Operating Systems Review* 44, 4 (Dec. 2010), 124–135.

[12] Bell, D. E. and Lapadula, L. J. *Secure Computer Systems: Mathematical Foundations.* Tech. rep. 2547. MITRE, 1973.

[13] Biba, K. J. *Integrity Considerations for Secure Computer Systems.* Tech. rep. ESD-TR-76-372. USAF Electronic Sstems Division, Hanscom Air Force Base, 1977.

[14] Bickford, J., O'Hare, R., Baliga, A., Ganapathy, V., and Iftode, L. Rootkits on Smart Phones: Attacks, Implications and Opportunities. In: *Proceedings of the 11th Workshop on Mobile Computing Systems and Applications (HotMobile 2010).* ACM, 2010.

[15] Bleikertz, S. and Gross, T. A Virtualization Assurance Language for Isolation and Deployment. In: *Proceedings of the 2011 IEEE International Symposium on Policies for Distributed Systems and Networks (POLICY 2011).* IEEE, 2011.

[16] Bradley, T. *DroidDream Becomes Android Market Nightmare.* Online: http://www.pcworld.com/businesscenter/article/221247/droiddream_becomes_android_market_nightmare.html. 2011.

[17] BRADY, P. *Anatomy & Physiology of an Android (2008 Google I/O Session Videos and Slides).* Online: `https://sites.google.com/site/io/anatomy--physiology-of-an-android`. 2008.

[18] BREWER, D. F. and NASH, M. J. The Chinese Wall Security Policy. In: *Proceedings of the 10th IEEE Symposium on Security and Privacy (Oakland 1989).* IEEE, 1989.

[19] CAI, L. and CHEN, H. TouchLogger: inferring keystrokes on touch screen from smartphone motion. In: *Proceedings of the 6th USENIX Workshop on Hot Topics in Security (HotSec'11).* USENIX Association, 2011.

[20] CARTER, J. *Using GConf as an Example of How to Create an Userspace Object Manager (SELinux Symposium).* 2007.

[21] CHIN, E., PORTER FELT, A., GREENWOOD, K., and WAGNER, D. Analyzing inter-application communication in Android. In: *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services (MobiSys 2011).* ACM, 2011.

[22] CLARK, D. D. and WILSON, D. R. A Comparison of Commercial and Military Computer Security Policies. In: *Proceedings of the 8th IEEE Symposium on Security and Privacy (Oakland 1987).* IEEE, 1987.

[23] *Contagio Mobile.* Online: `http://contagiominidump.blogspot.de/`.

[24] CONTI, M., NGUYEN, V. T. N., and CRISPO, B. CRePE: Context-Related Policy Enforcement for Android. In: *Proceedings of the 13th Information Security Conference (ISC 2010).* Springer-Verlag, 2010.

[25] *CVE-2007-4993: Xen guest root can escape to Domain 0 through pygrub.* Online: `https://bugzilla.redhat.com/show_bug.cgi?id=CVE-2007-4993`. 2007.

[26] *CVE-2008-2100: VMWare buffer overflows in VIX API let local users execute arbitrary code.* Online: `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-2100`. 2008.

[27] *CVE-2012-3515: Device emulation in Qemu for Xen 4.0 and 4.1 allows local users to gain elevated privileges.* Online: `http://www.cvedetails.com/cve/CVE-2012-3515/`. 2012.

[28] *CVE-2014-1666: Insufficient restrictions in physical device operations in multiple Xen version allow local guests to gain elevated privileges.* Online: `http://www.cvedetails.com/cve/CVE-2014-1666/`. 2014.

[29] DAISUKE, N. and TONA, G. L. *Tomoyo-android: TOMOYO Linux on Android.* Online: `http://code.google.com/p/tomoyo-android/`.

[30] DAVI, L., DMITRIENKO, A., SADEGHI, A.-R., and WINANDY, M. Privilege Escalation Attacks on Android. In: *Proceedings of the 13th Information Security Conference (ISC 2010).* Springer-Verlag, 2010.

[31] Davi, L., Dmitrienko, A., Kowalski, C., and Winandy, M. Trusted Virtual Domains on OKL4: Secure Information Sharing on Smartphones. In: *Proceedings of the 6th Annual Workshop on Scalable Trusted Computing (STC 2011)*. ACM, 2011.

[32] David, F., Chan, E., Carlyle, J., and Campbell, R. Cloaker: Hardware Supported Rootkit Concealment. In: *Proceedings of the 29th IEEE Symposium on Security and Privacy (Oakland 2008)*. IEEE, 2008.

[33] Davis, B. and Chen, H. RetroSkeleton: Retrofitting Android Apps. In: *Proceedings of the 11th International Conference on Mobile Systems, Applications, and Services (MobiSys 2013)*. ACM, 2013.

[34] Davis, B., Sanders, B., Khodaverdian, A., and Chen, H. I-ARM-Droid: A Rewriting Framework for In-App Reference Monitors for Android Applications. In: *Proceedings of the 2012 Mobile Security Technologies Workshop (MoST 2012)*. IEEE, 2012.

[35] Denning, D. E. A Lattice Model of Secure Information Flow. *Communications of the ACM* 19, 5 (May 1976), 236–243.

[36] Dietz, M., Shekhar, S., Pisetsky, Y., Shu, A., and Wallach, D. S. QUIRE: Lightweight Provenance for Smart Phone Operating Systems. In: *Proceedings of the 20th Usenix Security Symposium (SEC 2011)*. USENIX Association, 2011.

[37] Distefano, A., Grillo, A., Lentini, A., and Italiano, G. F. SecureMyDroid: enforcing security in the mobile devices lifecycle. In: *Proceedings of the 6th ACM Annual Workshop on Cyber Security and Information Intelligence Research (CSIIRW 2010)*. ACM, 2010.

[38] Dmitrienko, A., Eriksson, K., Kuhlmann, D., Ramunno, G., Sadeghi, A.-R., Schulz, S., Schunter, M., Winandy, M., Catuogno, L., and Zhan, J. Trusted Virtual Domains – Design, Implementation and Lessons Learned. In: *Proceedings of the 1st Conference on Trusted Systems (INTRUST 2009)*. Springer-Verlag, 2009.

[39] Dolzhenko, E., Ligatti, J., and Reddy, S. Modeling runtime enforcement with mandatory results automata. *International Journal of Information Security* 14, 1 (2015), 47–60.

[40] Edge, J. *The return of loadable security modules?* Online: `http://lwn.net/Articles/526983/`. Nov. 2012.

[41] Edwards, A., Jaeger, T., and Zhang, X. Runtime Verification of Authorization Hook Placement for the Linux Security Modules Framework. In: *Proceedings of the 9th ACM Conference on Computer and Communication Security (CCS 2002)*. ACM, 2002.

[42] Efstathopoulos, P., Krohn, M., VanDeBogart, S., Frey, C., Ziegler, D., Kohler, E., Mazières, D., Kaashoek, F., and Morris, R. Labels and event processes in the Asbestos operating system. In: *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP 2005)*. ACM, 2005.

[43] Elish, K., Yao, D., and Ryder, B. On the Need of Precise Inter-App ICC Classification for Detecting Android Malware Collusions. In: *Proceedings of the 2015 Mobile Security Technologies Workshop (MoST 2015)*. IEEE, 2015.

[44] Enck, W., Ongtang, M., and McDaniel, P. On lightweight mobile phone application certification. In: *Proceedings of the 16th ACM Conference on Computer and Communication Security (CCS 2009)*. ACM, 2009.

[45] Enck, W., Ongtang, M., and McDaniel, P. Understanding Android Security. *IEEE Security and Privacy* 7, 1 (2009), 50–57.

[46] Enck, W., Gilbert, P., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P., and Sheth, A. N. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In: *Proceedings of the 9th Usenix Symposium on Operating Systems Design and Implementation (OSDI 2010)*. USENIX Association, 2010.

[47] Enck, W., Octeau, D., McDaniel, P., and Chaudhuri, S. A Study of Android Application Security. In: *Proceedings of the 20th Usenix Security Symposium (SEC 2011)*. USENIX Association, 2011.

[48] Erlingsson, Ú. The Inlined Reference Monitor Approach to Security Policy Enforcement. PhD thesis. Cornell University, Jan. 2004.

[49] Erlingsson, Ú. and Schneider, F. B. IRM Enforcement of Java Stack Inspection. In: *Proceedings of the 21st IEEE Symposium on Security and Privacy (Oakland 2000)*. IEEE, 2000.

[50] F-Secure Labs. *Mobile Threat Report: Q3 2012*. 2012.

[51] *Facebook Caught Reading User SMS Messages? | TalkAndroid.com*. `http://www.talkandroid.com/94623-facebook-caught-reading-user-sms-messages/`. 2012.

[52] Fahl, S., Harbach, M., Oltrogge, M., Muders, T., and Smith, M. Hey, You, Get Off of My Clipboard. In: *Proceedings of 2013 Financial Cryptography and Data Security (FC 2013)*. Springer-Verlag, 2013.

[53] Fähndrich, M., Aiken, M., Hawblitzel, C., Hodson, O., Hunt, G., Larus, J. R., and Levi, S. Language support for fast and reliable message-based communication in Singularity OS. In: *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys 2006)*. ACM, 2006.

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

[55] Ford, B. and Lepreau, J. Evolving Mach 3.0 to a migrating thread model. In: *Proceedings of the USENIX Winter 1994 Technical Conference (WTEC 1994)*. USENIX Association, 1994.

[56] Fraser, T., Badger, L., and Feldman, M. Hardening COTS software with generic software wrappers. In: *Proceedings of the 20th IEEE Symposium on Security and Privacy (Oakland 1999)*. IEEE, 1999.

[57] Fraser, T. LOMAC: Low Water-Mark Integrity Protection for COTS Environments. In: *Proceedings of the 21st IEEE Symposium on Security and Privacy (Oakland 2000)*. IEEE, 2000.

[58] Fraser, T. LOMAC: MAC You Can Live With. In: *Proceedings of the 2001 USENIX Annual Technical Conference (ATC 2001)*. USENIX Association, 2001.

[59] Fratantonio, Y., Bianchi, A., Robertson, W., Egele, M., Kruegel, C., Kirda, E., and Vigna, G. On the Security and Engineering Implications of Finer-Grained Access Controls for Android Developers and Users. In: *Proceedings of the 12th Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA 2015)*. Springer-Verlag, 2015.

[60] Ganapathy, V., Jaeger, T., and Jha, S. Automatic Placement of Authorization Hooks in the Linux Security Modules Framework. In: *Proceedings of the 12th ACM Conference on Computer and Communication Security (CCS 2005)*. ACM, 2005.

[61] Ganapathy, V., Jaeger, T., and Jha, S. Retrofitting Legacy Code for Authorization Policy Enforcement. In: *Proceedings of the 27th IEEE Symposium on Security and Privacy (Oakland 2006)*. IEEE, 2006.

[62] Gartner. *Gartner Says Smartphone Sales Surpassed One Billion Units in 2014*. Online: http://www.gartner.com/newsroom/id/2996817. Mar. 2015.

[63] Ge, X., Vijayakumar, H., and Jaeger, T. Sprobes: Enforcing Kernel Code Integrity on the TrustZone Architecture. In: *Proceedings of the 2014 Mobile Security Technologies Workshop (MoST 2014)*. IEEE, 2014.

[64] Gehani, A. and Tariq, D. SPADE: Support for Provenance Auditing in Distributed Environments. In: *Middleware 2012*. Vol. 7662. Lecture Notes in Computer Science. Springer-Verlag, 2012.

[65] Gilbert, P., Chun, B.-G., Cox, L., and Jung, J. *Automating Privacy Testing of Smartphone Applications*. Tech. rep. CS-2011-02. Duke University, 2011.

[66] Gilbert, P., Chun, B.-G., Cox, L. P., and Jung, J. Vision: automated security validation of mobile apps at app markets. In: *Proceedings of the 2nd International Workshop on Mobile Cloud Computing and Services (MCS 2011)*. ACM, 2011.

[67] Gligor, V., Gavrila, S., and Ferraiolo, D. On the formal definition of separation-of-duty policies and their composition. In: *Proceedings of the 19th IEEE Symposium on Security and Privacy (Oakland 1998)*. IEEE, 1998.

[68] Goguen, J. A. and Meseguer, J. Security Policies and Security Models. In: *Proceedings of the 3rd IEEE Symposium on Security and Privacy (Oakland 1982)*. IEEE, 1982.

[69] Google. *Enhancing Security with Device Management Policies*. Online: http://developer.android.com/training/enterprise/device-management-policy.html.

[70] Google. *Nexus Help: Restricted profiles*. Online: https://support.google.com/nexus/answer/3175031?hl=en.

[71] Google. *Nexus Help: Use screen pinning.* Online: https://support.google.com/nexus/answer/6118421?hl=en.

[72] Google. *Security-Enhanced Linux in Android.* Online: https://source.android.com/devices/tech/security/selinux/index.html.

[73] Google. *Security Enhancements in Android 4.2.* Online: http://source.android.com/devices/tech/security/enhancements42.html.

[74] Google. *Security Enhancements in Android 4.3.* Online: http://source.android.com/devices/tech/security/enhancements43.html.

[75] Google. *Security Enhancements in Android 4.4.* Online: http://source.android.com/devices/tech/security/enhancements44.html.

[76] Google. *Security Enhancements in Android 5.0.* Online: https://source.android.com/devices/tech/security/enhancements/enhancements50.html.

[77] Google. *System Permissions: URI Permissions.* Online: http://developer.android.com/guide/topics/security/permissions.html#uri.

[78] Google. *Android Auto.* Online: https://developer.android.com/auto/index.html. 2015.

[79] Google. *Android: Codenames, Tags, and Build Numbers.* Online: https://source.android.com/source/build-numbers.html. 2015.

[80] Google. *Android for Work: Security white paper.* May 2015.

[81] Google. *Android TV.* Online: http://www.android.com/tv/. 2015.

[82] Google. *Android Wear.* Online: https://developer.android.com/wear/index.html. 2015.

[83] Google. *Project Brillo.* Online: https://developers.google.com/brillo/. 2015.

[84] Grace, M., Zhou, W., Jiang, X., and Sadeghi, A.-R. 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 2012).* ACM, 2012.

[85] Guttman, J. D., Herzog, A. L., Ramsdell, J. D., and Skorupka, C. W. Verifying Information Flow Goals in Security-enhanced Linux. *Journal of Computer Security* 13, 1 (Jan. 2005), 115–134.

[86] Hao, H., Singh, V., and Du, W. On the Effectiveness of API-level Access Control Using Bytecode Rewriting in Android. In: *Proceedings of the 8th ACM Symposium on Information, Computer and Communication Security (ASIACCS 2013).* ACM, 2013.

[87] Harada, T., Horie, T., and Tanaka, K. Task Oriented Management Obviates Your Onus on Linux. In: *Linux Conference.* 2004.

[88] Hardy, N. The Confused Deputy: (or why capabilities might have been invented). *SIGOPS Operating Systems Review* 22, 4 (Oct. 1988), 36–38.

[89] Harrison, M. A., Ruzzo, W. L., and Ullman, J. D. Protection in operating systems. *Communications of the ACM* 19, 8 (Aug. 1976), 461–471. issn: 0001-0782.

[90] Heuser, S., Nadkarni, A., Enck, W., and Sadeghi, A.-R. ASM: A Programmable Interface for Extending Android Security. In: *Proceedings of the 23rd USENIX Security Symposium (SEC 2014)*. USENIX Association, 2014.

[91] Hicks, B., King, D., McDaniel, P., and Hicks, M. Trusted Declassification:: High-level Policy for a Security-typed Language. In: *Proceedings of the 1st ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS 2006)*. ACM, 2006.

[92] Hicks, B., Rueda, S., Jaeger, T., and McDaniel, P. From Trusted to Secure: Building and Executing Applications That Enforce System Security. In: *Proceedings of the 2007 USENIX Annual Technical Conference (ATC 2007)*. USENIX Association, 2007.

[93] Hornyack, P., Han, S., Jung, J., Schechter, S., and Wetherall, D. 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 Communication Security (CCS 2011)*. ACM, 2011.

[94] Hu, H., Ahn, G.-J., and Kulkarni, K. Detecting and Resolving Firewall Policy Anomalies. *IEEE Transactions on Dependable and Secure Computing* 9, 3 (May 2012), 318–331.

[95] Husted, N., Qureshi, S., Tariq, D., and Gehani, A. Android provenance: diagnosing device disorders. In: *Proceedings of the 5th USENIX Workshop on the Theory and Practice of Provenance (TaPP 2015)*. USENIX Association, 2013.

[96] Hwang, J.-Y., Suh, S.-B., Heo, S.-K., Park, C.-J., Ryu, J.-M., Park, S.-Y., and Kim, C.-R. 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 2008)*. IEEE, 2008.

[97] Jaeger, T. Managing Access Control Complexity Using Metrics. In: *Proceedings of the 6th ACM Symposium on Access Control Models and Technologies (SACMAT 2001)*. ACM, 2001.

[98] Jaeger, T., Liedtke, J., and Islam, N. Operating System Protection for Fine-grained Programs. In: *Proceedings of the 7th Usenix Security Symposium (SEC 1998)*. USENIX Association, 1998.

[99] Jaeger, T., Sailer, R., and Shankar, U. PRIMA: Policy-reduced Integrity Measurement Architecture. In: *Proceedings of the 11th ACM Symposium on Access Control Models and Technologies (SACMAT 2006)*. ACM, 2006.

[100] Jaeger, T., Sailer, R., and Zhang, X. Analyzing Integrity Protection in the SELinux Example Policy. In: *Proceedings of the 12th Usenix Security Symposium (SEC 2003)*. USENIX Association, 2003.

[101] Jeon, J., Micinski, K. K., Vaughan, J. A., Fogel, A., Reddy, N., Foster, J. S., and Millstein, T. Dr. Android and Mr. Hide: Fine-grained security policies on unmodified Android. In: *Proceedings of the 2nd ACM CCS Workshop on Security and Privacy in Mobile Devices (SPSM 2012)*. ACM, 2012.

[102] Kantola, D., Chin, E., He, W., and Wagner, D. Reducing attack surfaces for intra-application communication in Android. In: *Proceedings of the 2nd ACM CCS Workshop on Security and Privacy in Mobile Devices (SPSM 2012)*. ACM, 2012.

[103] Kennedy, K., Gustafson, E., and Chen, H. Quantifying the Effects of Removing Permissions from Android Applications. In: *Proceedings of the 2013 Mobile Security Technologies Workshop (MoST 2013)*. IEEE, 2013.

[104] Kohel, K. *Security Enhanced PostgreSQL (SELinux Symposium)*. 2007.

[105] Kostiainen, K., Reshetova, E., Ekberg, J.-E., and Asokan, N. Old, new, borrowed, blue – a perspective on the evolution of mobile platform security architectures. In: *Proceedings of the 1st ACM Conference on Data and Application Security and Privacy (CODASPY 2011)*. ACM, 2011.

[106] Lampson, B. W. Protection. *ACM SIGOPS Operating Systems Review* 8, 1 (Jan. 1974), 18–24.

[107] Lampson, B. W. and Sturgis, H. E. Reflections on an operating system design. *Communications of the ACM* 19, 5 (May 1976), 251–265.

[108] Lange, M., Liebergeld, S., Lackorzynski, A., Warg, A., and Peter, M. L4Android: A Generic Operating System Framework for Secure Smartphones. In: *Proceedings of the 1st ACM CCS Workshop on Security and Privacy in Mobile Devices (SPSM 2011)*. ACM, 2011.

[109] Ligatti, J., Bauer, L., and Walker, D. Edit Automata: Enforcement Mechanisms for Run-time Security Policies. *International Journal of Information Security* 4, 1–2 (2005), 2–16.

[110] Linden, T. A. Operating System Structures to Support Security and Reliable Software. *ACM Computer Surveys* 8, 4 (Dec. 1976), 409–445.

[111] Lineberry, A., Richardson, D. L., and Wyatt, T. *These aren't the permissions you're looking for. BlackHat USA 2010*. Online: `http://dtors.files.wordpress.com/2010/08/blackhat-2010-slides.pdf`. 2010.

[112] Linux Cross Reference. *Linux Security Module framework*. Online: `http://lxr.free-electrons.com/source/Documentation/security/LSM.txt`.

[113] Lookout Mobile Security. *Security Alert: Geinimi, Sophisticated New Android Trojan Found in Wild*. Online: `http://blog.mylookout.com/2010/12/geinimi_trojan/`. 2010.

[114] Loscocco, P. and Smalley, S. Integrating Flexible Support for Security Policies into the Linux Operating System. In: *Proceedings of the 2001 USENIX Annual Technical Conference (ATC 2001)*. USENIX Association, 2001.

[115] Loscocco, P. A., Smalley, S. D., Muckelbauer, P. A., Taylor, R. C., Turner, S. J., and Farrell, J. F. The Inevitability of Failure: The Flawed Assumption of Security in Modern Computing Environments. In: *Proceedings of the 21st National Information Systems Security Conference (NISSC 1998).* NIST, 1998.

[116] Mallempati, R. *Google I/O Recap, Part 1: Google is Serious About Enterprise Mobility.* Online: `https://www.mobileiron.com/en/smartwork-blog/google-io-recap-part-1-google-serious-about-enterprise-mobility`. June 2014.

[117] Marforio, C., Ritzdorf, H., Francillon, A., and Čapkun, S. Analysis of the communication between colluding applications on modern smartphones. In: *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC 2012).* ACM, 2012.

[118] Mayer, F., MacMillan, K., and Caplan, D. *SELinux by Example: Using Security Enhanced Linux (Prentice Hall Open Source Software Development Series).* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2006. isbn: 0131963694.

[119] McAfee. *Virus Profile: Android/Loozfon.A.* Online: `http://home.mcafee.com/virusinfo/virusprofile.aspx?key=2316508`.

[120] McDaniel, P. and Prakash, A. Methods and limitations of security policy reconciliation. In: *Proceedings of the 23rd IEEE Symposium on Security and Privacy (Oakland 2002).* IEEE, 2002.

[121] McDaniel, P., Butler, K., McLaughlin, S., Sion, R., Zadok, E., and Winslett, M. Towards a secure and efficient system for end-to-end provenance. In: *Proceedings of the 2nd USENIX Workshop on the Theory and Practice of Provenance (TaPP 2010).* USENIX Association, 2010.

[122] Miettinen, M., Heuser, S., Kronz, W., Sadeghi, A.-R., and Asokan, N. ConXsense: Automated Context Classification for Context-aware Access Control. In: *Proceedings of the 9th ACM Symposium on Information, Computer and Communication Security (ASIACCS 2014).* ACM, 2014.

[123] Migliavacca, M., Papagiannis, I., Eyers, D. M., Shand, B., Bacon, J., and Pietzuch, P. DEFCON: high-performance event processing with information security. In: *Proceedings of the 2010 USENIX Annual Technical Conference (ATC 2010).* USENIX Association, 2010.

[124] Morrill, D. Online: `http://android-developers.blogspot.de/2008/09/announcing-android-10-sdk-release-1.html`. Sept. 2008.

[125] Moulu, A. *Android OEM's applications (in)security and backdoors without permission.* Online: `http://www.quarkslab.com/dl/Android-OEM-applications-insecurity-and-backdoors-without-permission.pdf`.

[126] Muthukumaran, D., Schiffman, J., Hassan, M., Sawani, A., Rao, V., and Jaeger, T. Protecting the Integrity of Trusted Applications in Mobile Phone Systems. *Security and Communication Networks* 4, 6 (2011), 633–650.

[127] Muthukumaran, D., Jaeger, T., and Ganapathy, V. Leveraging "Choice" to Automate Authorization Hook Placement. In: *Proceedings of the 19th ACM Conference on Computer and Communication Security (CCS 2012)*. ACM, 2012.

[128] Muthukumaran, D., Sawani, A., Schiffman, J., Jung, B. M., and Jaeger, T. Measuring Integrity on Mobile Phone Systems. In: *Proceedings of the 13th ACM Symposium on Access Control Models and Technologies (SACMAT 2008)*. ACM, 2008.

[129] Myers, A. C. and Liskov, B. A decentralized model for information flow control. In: *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP 1997)*. ACM, 1997.

[130] Myers, A. C. and Liskov, B. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 9, 4 (Oct. 2000), 410–442.

[131] Nadkarni, A. and Enck, W. Preventing Accidental Data Disclosure in Modern Operating Systems. In: *Proceedings of the 20th ACM Conference on Computer and Communication Security (CCS 2013)*. ACM, 2013.

[132] National Security Agency. *Security Enhancements (SE) for Android*. Online: `http://seandroid.bitbucket.org`.

[133] Nauman, M., Khan, S., and Zhang, X. Apex: Extending Android permission model and enforcement with user-defined runtime constraints. In: *Proceedings of the 5th ACM Symposium on Information, Computer and Communication Security (ASIACCS 2010)*. ACM, 2010.

[134] Nauman, M., Khan, S., Zhang, X., and Seifert, J.-P. Beyond kernel-level integrity measurement: Enabling remote attestation for the Android platform. In: *Proceedings of the 3rd International Conference on Trust and Trustworthy Computing (TRUST 2010)*. Springer-Verlag, 2010.

[135] Nils. *Building Android Sandcastles in Android's Sandbox. (BlackHat Abu Dhabi 2010)*. Online: `https://media.blackhat.com/bh-ad-10/Nils/Black-Hat-AD-2010-android-sandcastle-wp.pdf`. 2010.

[136] Oberheide, J. *Android Hax. SummerCon 2010*. Online: `http://jon.oberheide.org/files/summercon10-androidhax-jonoberheide.pdf`. 2010.

[137] Octeau, D., McDaniel, P., Jha, S., Bartel, A., Bodden, E., Klein, J., and Traon, Y. L. Effective Inter-Component Communication Mapping in Android: An Essential Step Towards Holistic Security Analysis. In: *Proceedings of the 22nd Usenix Security Symposium (SEC 2013)*. USENIX Association, 2013.

[138] Oluwafemi, T., Fernandes, E., Riva, O., Roesner, F., Nath, S., and Kohno, T. *Per-App Profiles with AppFork: The Security of Two Phones with the Convenience of One*. Tech. rep. MSR-TR-2014-153. Dec. 2014.

[139] Ongtang, M., Butler, K., and McDaniel, P. Porscha: Policy Oriented Secure Content Handling in Android. In: *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC 2010)*. ACM, 2010.

[140] Ongtang, M., McLaughlin, S. E., Enck, W., and McDaniel, P. Semantically Rich Application-Centric Security in Android. In: *Proceedings of the 25th Annual Computer Security Applications Conference (ACSAC 2009)*. ACM, 2009.

[141] Ott, A. The Rule Set Based Access Control (RSBAC) Linux Kernel Security Extension. In: *8th International Linux Kongress*. 2001.

[142] Palm Source, Inc. *Open Binder. Version 1*. Online: `http://www.angryredplanet.com/~hackbod/openbinder/docs/html/index.html`. 2005.

[143] Palm Source, Inc. *Open Binder, Version 1.0*. `http://www.angryredplanet.com/~hackbod/openbinder/docs/html/`. 2005.

[144] Pearce, P., Porter Felt, A., Nunez, G., and Wagner, D. AdDroid: Privilege Separation for Applications and Advertisers in Android. In: *Proceedings of the 7th ACM Symposium on Information, Computer and Communication Security (ASIACCS 2012)*. ACM, 2012.

[145] Perez, S. *Security Hole Spotted In Facebook Android SDK, Long Tail Apps May Still Be Unpatched*. Online: `http://techcrunch.com/2012/04/10/security-hole-spotted-in-facebook-android-sdk-long-tail-apps-may-still-be-unpatched/`. Apr. 2012.

[146] Porter Felt, A., Finifter, M., Chin, E., Hanna, S., and Wagner, D. A survey of mobile malware in the wild. In: *Proceedings of the 1st ACM CCS Workshop on Security and Privacy in Mobile Devices (SPSM 2011)*. ACM, 2011.

[147] Porter Felt, A., Chin, E., Hanna, S., Song, D., and Wagner, D. Android Permissions Demystified. In: *Proceedings of the 18th ACM Conference on Computer and Communication Security (CCS 2011)*. ACM, 2011.

[148] Porter Felt, A., Wang, H. J., Hanna, A. M. andSteve, and Chin, E. Permission Re-Delegation: Attacks and Defenses. In: *Proceedings of the 20th Usenix Security Symposium (SEC 2011)*. USENIX Association, 2011.

[149] Portokalidis, G., Homburg, P., Anagnostakis, K., and Bos, H. Paranoid Android: Versatile Protection For Smartphones. In: *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC 2010)*. ACM, 2010.

[150] Provos, N. Improving host security with system call policies. In: *Proceedings of the 12th Usenix Security Symposium (SEC 2003)*. USENIX Association, 2003.

[151] Provos, N., Friedl, M., and Honeyman, P. Preventing Privilege Escalation. In: *Proceedings of the 12th Usenix Security Symposium (SEC 2003)*. USENIX Association, 2003.

[152] Rao, V. and Jaeger, T. Dynamic mandatory access control for multiple stakeholders. In: *Proceedings of the 14th ACM Symposium on Access Control Models and Technologies (SACMAT 2009)*. ACM, 2009.

[153] Ratazzi, P., Bommisetti, A., Ji, N., and Du, W. PINPOINT: Efficient and Effective Resource Isolation for Mobile Security and Privacy. In: *Proceedings of the 2015 Mobile Security Technologies Workshop (MoST 2015)*. IEEE, 2015.

[154] Reeder, R. W., Bauer, L., Cranor, L. F., Reiter, M. K., and Vaniea, K. More than skin deep: measuring effects of the underlying model on access-control system usability. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI 2011)*. ACM, 2011.

[155] Rushby, J. M. Design and Verification of Secure Systems. In: *Proceedings of the 8th ACM Symposium on Operating Systems Principles (SOSP 1981)*. ACM, 1981.

[156] Russello, G., Conti, M., Crispo, B., and Fernandes, E. MOSES: supporting operation modes on smartphones. In: *Proceedings of the 17th ACM Symposium on Access Control Models and Technologies (SACMAT 2012)*. ACM, 2012.

[157] Sailer, R., Jaeger, T., Zhang, X., and Doorn, L. van. Attestation-based Policy Enforcement for Remote Access. In: *Proceedings of the 11th ACM Conference on Computer and Communication Security (CCS 2004)*. ACM, 2004.

[158] Saltzer, J. and Schroeder, M. The protection of information in computer systems. *Proceedings of the IEEE* 63, 9 (1975), 1278–1308.

[159] Samsung. *Knox*. Online: `https://www.samsungknox.com`.

[160] Samsung. *Galaxy Cameras*. Online: `http://www.samsung.com/us/photography/galaxy-camera`. 2015.

[161] Schlegel, R., Zhang, K., Zhou, X., Intwala, M., Kapadia, A., and Wang, X. Soundcomber: A Stealthy and Context-Aware Sound Trojan for Smartphones. In: *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS 2011)*. The Internet Society, 2011.

[162] Schreiber, T. *Android Binder – Android Interprocess Communication*. `https://www.nds.rub.de/media/attachments/files/2011/10/main.pdf`. 2011.

[163] Schroeder, M. D., Clark, D. D., and Saltzer, J. H. The Multics kernel design project. In: *Proceedings of the 6th ACM Symposium on Operating Systems Principles (SOSP 1977)*. ACM, 1977.

[164] Sehr, D., Muth, R., Biffle, C., Khimenko, V., Pasko, E., Schimpf, K., Yee, B., and Chen, B. Adapting software fault isolation to contemporary CPU architectures. In: *Proceedings of the 19th Usenix Security Symposium (SEC 2010)*. USENIX Association, 2010.

[165] Selhorst, M., Stüble, C., Feldmann, F., and Gnaida, U. Towards a trusted mobile desktop. In: *Proceedings of the 3rd International Conference on Trust and Trustworthy Computing (TRUST 2010)*. Springer-Verlag, 2010.

[166] Shabtai, A., Fledel, Y., and Elovici, Y. Securing Android-Powered Mobile Devices Using SELinux. *IEEE Security Privacy* 8, 3 (May 2010), 36–44.

[167] Shankar, U., Jaeger, T., and Sailer, R. Toward Automated Information-Flow Integrity Verification for Security-Critical Applications. In: *Proceedings of the 13th Annual Network and Distributed System Security Symposium (NDSS 2006)*. The Internet Society, 2006.

[168] Shapiro, J. S. Vulnerabilities in Synchronous IPC Designs. In: *Proceedings of the 24th IEEE Symposium on Security and Privacy (Oakland 2003)*. IEEE, 2003.

[169] Shapiro, J. S., Smith, J. M., and Farber, D. J. EROS: a fast capability system. In: *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP 1999)*. ACM, 1999.

[170] Shekhar, S., Dietz, M., and Wallach, D. S. AdSplit: Separating Smartphone Advertising from Applications. In: *Proceedings of the 21st Usenix Security Symposium (SEC 2012)*. USENIX Association, 2012.

[171] Simmhan, Y. L., Plale, B., and Gannon, D. A survey of data provenance in e-science. *SIGMOD Record* 34, 3 (Sept. 2005), 31–36.

[172] Smalley, S. *Middleware MAC for Android (Linux Security Summit 2012)*. Online: `http://kernsec.org/files/LSS2012-MiddlewareMAC.pdf`. Aug. 2012.

[173] Smalley, S. *Security Enhanced (SE) Android (Presentation)*. Aug. 2012.

[174] Smalley, S. *The Case for SE Android (Linux Security Summit 2011)*. Online: `http://www.selinuxproject.org/~jmorris/lss2011_slides/caseforseandroid.pdf`. 2012.

[175] Smalley, S. and Craig, R. Security Enhanced (SE) Android: Bringing Flexible MAC to Android. In: *Proceedings of the 20th Annual Network and Distributed System Security Symposium (NDSS 2013)*. The Internet Society, 2013.

[176] Smalley, S., Vance, C., and Salamon, W. *Implementing SELinux as a Linux Security Module*. NAI Labs Report #01-043. Revised May 2002. NAI Labs, Dec. 2001.

[177] Smith, C. *Privacy flaw in Skype Android app exposed*. Online: `http://www.t3.com/news/privacy-flaw-in-skype-android-app-exposed/`. 2011.

[178] Sniffen, B. T., Harris, D. R., and Ramsdell, J. D. *Guided Policy Generation for Application Authors (SELinux Symposium)*. 2006.

[179] Spencer, R., Smalley, S., Loscocco, P., Hibler, M., Andersen, D., and Lepreau, J. The Flask Security Architecture: System Support for Diverse Security Policies. In: *Proceedings of the 8th Usenix Security Symposium (SEC 1999)*. USENIX Association, 1999.

[180] Srage, J. and Azema, J. *M-Shield Mobile Security Technology*. TI White paper. Online: `http://focus.ti.com/pdfs/wtbu/ti_mshield_whitepaper.pdf`. 2005.

[181] Sun, M. and Tan, G. 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 2014)*. ACM, 2014.

[182] Symantec. *Android.Enesoluty*. Online: `http://www.symantec.com/security_response/writeup.jsp?docid=2012-090607-0807-99`.

[183] Templeman, R., Rahman, Z., Crandall, D., and Kapadia, A. PlaceRaider: Virtual Theft in Physical Spaces with Smartphones. In: *Proceedings of the 20th Annual Network and Distributed System Security Symposium (NDSS 2013)*. The Internet Society, 2013.

[184] *TOMOYO Linux Wiki: How is TOMOYO Linux different from SELinux and AppArmor?* Online: `http://tomoyo.sourceforge.jp/wiki-e/?WhatIs#comparison`.

[185] Traynor, P., Lin, M., Ongtang, M., Rao, V., Jaeger, T., McDaniel, P., and La Porta, T. On Cellular Botnets: Measuring the Impact of Malicious Devices on a Cellular Network Core. In: *Proceedings of the 16th ACM Conference on Computer and Communication Security (CCS 2009)*. ACM, 2009.

[186] TrendMicro. *Android Under Siege: Popularity Comes at a Price.* 2012.

[187] Trusted Computing Group (TCG). *Mobile Trusted Module (MTM) Specification, Version 1.0 Revision 6.* 2008.

[188] Trusted Computing Group (TCG). *TNC Architecture for Interoperability, Version 1.4, Revision 4.* 2009.

[189] Vijayakumar, H., Ge, X., Payer, M., and Jaeger, T. JIGSAW: Protecting Resource Access by Inferring Programmer Expectations. In: *Proceedings of the 23rd USENIX Security Symposium (SEC 2014)*. USENIX Association, 2014.

[190] *Vulnerability in XenServer could result in privilege escalation and arbitrary code execution.* Online: `http://support.citrix.com/article/CTX118766`. 2008.

[191] Wallach, D. S. and Felten, E. W. Understanding Java Stack Inspection. In: *Proceedings of the 19th IEEE Symposium on Security and Privacy (Oakland 1998)*. IEEE, 1998.

[192] Walsh, E. *SELinux Support for Userspace Object Managers.* 2004.

[193] Walsh, E. *Application of the Flask Architecture to the X Window System Server.* 2007.

[194] Wang, R., Enck, W., Reeves, D., Zhang, X., Ning, P., Xu, D., Zhou, W., and Azab, A. EASEAndroid: Automatic Policy Analysis and Refinement for Security Enhanced Android via Large-Scale Semi-Supervised Learning. In: *Proceedings of the 24th USENIX Security Symposium (SEC 2015)*. USENIX Association, 2015.

[195] Wang, X., Sun, K., Wang, Y., and Jing, J. DeepDroid: Dynamically Enforcing Enterprise Policy on Android Devices. In: *Proceedings of the 22nd Annual Network and Distributed System Security Symposium (NDSS 2015)*. The Internet Society, 2015.

[196] Wang, Y., Hariharan, S., Zhao, C., Liu, J., and Du, W. Compac: Enforce Component-Level Access Control in Android. In: *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy (CODASPY 2014)*. ACM, 2014.

[197] Wang, Z. and Stavrou, A. Exploiting Smart-phone USB Connectivity for Fun and Profit. In: *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC 2010)*. ACM, 2010.

[198] Watson, R., Morrison, W., Vance, C., and Feldman, B. The TrustedBSD MAC Framework: Extensible Kernel Access Control for FreeBSD 5.0. In: *Proceedings of the 2003 USENIX Annual Technical Conference (ATC 2003)*. USENIX Association, 2003.

[199] Watson, R. N. M., Anderson, J., Laurie, B., and Kennaway, K. Capsicum: Practical Capabilities for UNIX. In: *Proceedings of the 19th Usenix Security Symposium (SEC 2010)*. USENIX Association, 2010.

[200] Watson, R. N. M., Woodruff, J., Neumann, P. G., Moore, S. W., Anderson, J., Chisnall, D., Dave, N., Davis, B., Gudka, K., Laurie, B., Murdoch, S. J., Norton, R., Roe, M., Son, S., and Vadera, M. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In: *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland 2015)*. IEEE, 2015.

[201] *WhatsApp storing messages of users up to 30 days | Your Daily Mac.* Online: `http://www.yourdailymac.net/2012/02/whatsapp-storing-messages-of-users-up-to-30-days/`. 2012.

[202] *WhatsApp took all my contacts and sent to their servers without asking me - BlackBerry Forums at CrackBerry.com.* Online: `http://forums.crackberry.com/blackberry-apps-f35/whatsapp-took-all-my-contacts-sent-their-servers-without-asking-me-649363/`. 2011.

[203] Winter, J. Trusted computing building blocks for embedded linux-based ARM trustzone platforms. In: *Proceedings of the 3rd Annual Workshop on Scalable Trusted Computing (STC 2008)*. ACM, 2008.

[204] Wobber, E., Abadi, M., Burrows, M., and Lampson, B. Authentication in the Taos Operating System. In: *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP 1993)*. ACM, 1993.

[205] Wojtczuk, R. and Rutkowska, J. *Xen 0wning Trilogy.* 2008.

[206] Wright, C., Cowan, C., Smalley, S., Morris, J., and Kroah-Hartman, G. Linux Security Modules: General Security Support for the Linux Kernel. In: *Proceedings of the 11th Usenix Security Symposium (SEC 2002)*. USENIX Association, 2002.

[207] Wu, C., Zhou, Y., Patel, K., Liang, Z., and Jiang, X. AirBag: Boosting Smartphone Resistance to Malware Infection. In: *Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS 2014)*. The Internet Society, 2014.

[208] *Xbox 360 Hypervisor Privilege Escalation Vulnerability.* Online: `http://www.securityfocus.com/archive/1/461489`. 2007.

[209] Xing, L., Pan, X., Wang, R., Yuan, K., and Wang, X. Upgrading Your Android, Elevating My Malware: Privilege Escalation Through Mobile OS Updating. In: *Proceedings of the 35th IEEE Symposium on Security and Privacy (Oakland 2014)*. IEEE, 2014.

[210] Xu, R., Saïdi, H., and Anderson, R. Aurasium: Practical Policy Enforcement for Android Applications. In: *Proceedings of the 21st Usenix Security Symposium (SEC 2012)*. USENIX Association, 2012.

[211] Xu, Z., Bai, K., and Zhu, S. 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 2012)*. ACM, 2012.

[212] Zhang, X., Seifert, J.-P., and Aciiçmez, O. SEIP: Simple and efficient integrity protection for open mobile platforms. In: *Proceedings of the 12th International Conference on Information and Communications Security (ICICS 2010)*. Springer-Verlag, 2010.

[213] Zhang, X., Aciiçmez, O., and Seifert, J.-P. A trusted mobile phone reference architecture via secure kernel. In: *Proceedings of the 2nd Annual Workshop on Scalable Trusted Computing (STC 2007)*. ACM, 2007.

[214] Zhou, X., Lee, Y., Zhang, N., Naveed, M., and Wang, X. The Peril of Fragmentation: Security Hazards in Android Device Driver Customizations. In: *Proceedings of the 35th IEEE Symposium on Security and Privacy (Oakland 2014)*. IEEE, 2014.

[215] Zhou, Y. and Jiang, X. Detecting Passive Content Leaks and Pollution in Android Applications. In: *Proceedings of the 20th Annual Network and Distributed System Security Symposium (NDSS 2013)*. The Internet Society, 2013.

[216] Zhou, Y., Wang, Z., Zhou, W., and Jiang, X. 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 2012)*. The Internet Society, 2012.

[217] Zhou, Y. and Jiang, X. Dissecting Android Malware: Characterization and Evolution. In: *Proceedings of the 33rd IEEE Symposium on Security and Privacy (Oakland 2012)*. IEEE, 2012.

[218] Zhou, Y., Zhang, X., Jiang, X., and Freeh, V. Taming Information-Stealing Smartphone Applications (on Android). In: *Proceedings of the 4th International Conference on Trust and Trustworthy Computing (TRUST 2011)*. Springer-Verlag, 2011.