



Saarland University
Faculty of Mathematics and Computer Science
Department of Computer Science

Retrofitting Privacy Controls to Stock Android

Dissertation
zur Erlangung des Grades
des Doktors der Ingenieurwissenschaften
der Fakultät für Mathematik und Informatik
der Universität des Saarlandes

von
Philipp von Styp-Rekowsky

Saarbrücken,
Dezember 2019

| | |
|---------------------------|---------------------------|
| Tag des Kolloquiums: | 18. Januar 2021 |
| Dekan: | Prof. Dr. Thomas Schuster |
| Prüfungsausschuss: | |
| Vorsitzender: | Prof. Dr. Thorsten Herfet |
| Berichterstattende: | Prof. Dr. Michael Backes |
| | Prof. Dr. Andreas Zeller |
| Akademischer Mitarbeiter: | Dr. Michael Schilling |

Zusammenfassung

Android ist nicht nur das beliebteste Betriebssystem für mobile Endgeräte, sondern auch ein attraktives Ziel für Angreifer. Um diesen zu begegnen, nutzt Androids Sicherheitskonzept App-Isolation und Zugangskontrolle zu kritischen Systemressourcen. Nutzer haben dabei aber nur wenige Optionen, App-Berechtigungen gemäß ihrer Bedürfnisse einzuschränken, sondern die Entwickler entscheiden über zu gewährende Berechtigungen. Androids Sicherheitsmodell kann zudem nicht durch Dritte angepasst werden, so dass Nutzer zum Schutz ihrer Privatsphäre auf die Gerätehersteller angewiesen sind. Diese Dissertation präsentiert einen Ansatz, Android mit umfassenden Privatsphäreinstellungen nachzurüsten. Dabei geht es konkret um Techniken, die ohne Modifikationen des Betriebssystems oder Zugriff auf Root-Rechte auf regulären Android-Geräten eingesetzt werden können. Der erste Teil dieser Arbeit etabliert Techniken zur Durchsetzung von Sicherheitsrichtlinien für Apps mithilfe von *inlined reference monitors*. Dieser Ansatz wird durch eine neue Technik für *dynamic method hook injection* in Androids Java VM erweitert. Schließlich wird ein System eingeführt, das prozessbasierte *privilege separation* nutzt, um eine virtualisierte App-Umgebung zu schaffen, um auch komplexe Sicherheitsrichtlinien durchzusetzen. Eine systematische Evaluation unseres Ansatzes konnte seine praktische Anwendbarkeit nachweisen und mehr als eine Million Downloads unserer Lösung zeigen den Bedarf an praxisgerechten Werkzeugen zum Schutz der Privatsphäre.

Abstract

Android is the most popular operating system for mobile devices, making it a prime target for attackers. To counter these, Android’s security concept uses app isolation and access control to critical system resources. However, Android gives users only limited options to restrict app permissions according to their privacy preferences but instead lets developers dictate the permissions users must grant. Moreover, Android’s security model is not designed to be customizable by third-party developers, forcing users to rely on device manufacturers to address their privacy concerns. This thesis presents a line of work that retrofits comprehensive privacy controls to the Android OS to put the user back in charge of their device. It focuses on developing techniques that can be deployed to stock Android devices without firmware modifications or root privileges. The first part of this dissertation establishes fundamental policy enforcement on third-party apps using inlined reference monitors to enhance Android’s permission system. This approach is then refined by introducing a novel technique for dynamic method hook injection on Android’s Java VM. Finally, we present a system that leverages process-based privilege separation to provide a virtualized application environment that supports the enforcement of complex security policies. A systematic evaluation of our approach demonstrates its practical applicability, and over one million downloads of our solution confirm user demand for privacy-enhancing tools.

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 idea for the initial work APPGUARD [P1, P2, T1, T2] and was responsible for its technical design, implementation and evaluation. Sebastian Gerling contributed the *SOSP*_X policy language to this work and developed further use-cases for APPGUARD. He was also involved in design discussions as well as the process of writing the paper. All authors performed reviews of the paper.

The method hooking technique presented in [P4] was developed by the author based on insights gained in our prior work on program instrumentation. While the author was solely responsible for the technical implementation of the idea, Sebastian Gerling and Christian Hammer contributed with general writing tasks. All authors performed reviews of the paper.

The initial idea for BOXIFY [P3] resulted from a discussion between the author and Sebastian Gerling following their joint work on APPGUARD. Again, the author was responsible for architecture, implementation and evaluation of BOXIFY, while Sven Bugiel contributed to the requirements analysis as well as to the writing progress. The paper was also reviewed by all the authors involved.

Author's Papers for this Thesis

- [P1] 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 2013)*. Springer-Verlag, 2013.
- [P2] BACKES, M., GERLING, S., HAMMER, C., MAFFEI, M., and STYP-REKOWSKY, P. von. AppGuard – Fine-grained Policy Enforcement for Untrusted Android Applications. In: *Proceedings of the 8th International Workshop on Data Privacy Management (DPM 2013)*. Springer-Verlag, 2013.
- [P3] 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.
- [P4] STYP-REKOWSKY, P. von, GERLING, S., BACKES, M., and HAMMER, C. Callee-site Rewriting of Sealed System Libraries. In: *Proceedings of the 5th International Symposium on Engineering Secure Software and Systems (ESSoS 2013)*. Springer-Verlag, 2013.

Further Contributions of the Author

- [S1] 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.

-
- [S2] BACKES, M., BUGIEL, S., SCHRANZ, O., STYP-REKOWSKY, P. von, and WEISSGERBER, S. ARTist: The Android runtime instrumentation and security toolkit. In: *Proceedings of the 2nd IEEE European Symposium on Security and Privacy (EuroS&P 2017)*. IEEE, 2017.
 - [S3] BACKES, M., BUGIEL, S., STYP-REKOWSKY, P. von, and WISSFELD, M. Seamless In-App Ad Blocking on Stock Android. In: *Proceedings of the 2017 Mobile Security Technologies Workshop (MoST 2017)*. IEEE, 2017.
 - [S4] JAMROZIK, K., STYP-REKOWSKY, P. von, and ZELLER, A. Mining Sandboxes. In: *Proceedings of the 38th International Conference on Software Engineering (ICSE 2016)*. ACM, 2016.

Technical Reports of the Author

- [T1] BACKES, M., GERLING, S., HAMMER, C., MAFFEI, M., and STYP-REKOWSKY, P. von. *AppGuard – Real-time policy enforcement for third-party applications*. Tech. rep. A/02/2012. Saarland University, 2012.
- [T2] BACKES, M., GERLING, S., HAMMER, C., MAFFEI, M., and STYP-REKOWSKY, P. von. *AppGuard – Fine-grained Policy Enforcement for Untrusted Android Applications*. Tech. rep. A/02/2013. Saarland University, 2013.
- [T3] 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, 2014.

Acknowledgments

First and foremost, I would like to thank my advisor Michael Backes: For the trust put in me, his patience and motivation, and his unwavering support throughout the writing of this thesis. Michael not only provided me with the opportunity to work in both academia and industry but also inspired me to think outside of the box and to explore new research directions. There also was an exciting period in early 2016, when we were in negotiations with a major tech company from the United States, which at that time showed interest in Boxify. I will personally never forget our long calls at the most unusual times where we discussed and refined our strategy. This experience is one that I do not want to miss and for which I am very thankful to Michael as well.

My sincere thanks also go to my long-time friend and university classmate Sebastian Gerling, who accompanied me throughout my studies from the very beginning and who first introduced me to Michael. It was also Sebastian, who initially pitched the idea to pursue a Ph.D. after my master's thesis. His structured and disciplined way to work proved incredibly helpful during our first years of university together, and the countless hours spent at his place doing exercise sheets not only paved the way for this thesis but also created a strong and valuable friendship.

I also want to thank my colleague Sven Bugiel, not only for his contributions to our research projects but also for our discussions about Android security in general. Sven's in-depth knowledge of the Android framework internals as well as his feedback and his advice that he was always happy to provide proved invaluable to the writing of this thesis. Mutual trips for work allowed me to get to know Sven's fun and social side, and I will fondly remember our matches of Cards against Humanity. I also thank my long-time office partner Erik Derr, who contributed to a workplace atmosphere that was enjoyable and often outright fun and who was always up for stimulating discussions. He also showed almost angelic patience with my collection of returnable bottles that only ever grew over the time we shared an office.

Many thanks also go to my closest collaborators and colleagues who contributed to this thesis in one way or the other. To these belong, in no particular order, Christian Hammer, Sven Obser, Fabian Bendun, Oliver Schranz, Sebastian Weisgerber, and Stefan Nürnberger. Together with Sebastian, Sven, and Erik, they all contributed to making my academic life fun and enjoyable, and we had lots of good times together outside of the office, be it at parties, over a beer at "Canossa" or other fun ventures like a round of laser-tag or karting. Our trips to Washington and San Francisco are also memories I fondly look back on when thinking about my years spent at Saarland University.

I also want to use this opportunity to thank Andreas Zeller for agreeing to be my examiner and for his ongoing encouragement and his insightful comments.

Last but not least, I want to thank my family and my friends, without whose support, love, and encouragement, this thesis would not have been possible. Special mention deserves Frederik Dressel for his help in finding the focus and motivation to conclude the writing down of this thesis and his invaluable literary support. I thank my parents, who instilled me with a love of science and who never ceased to support me in all the pursuits and endeavors that laid the groundwork for this Ph.D. thesis. Finally, I am especially grateful to my wife, Jasmin, who never ceases to provide me with moral and

emotional support, not only during the writing of this thesis but through my life as a whole.

Contents

| | | |
|----------|--|-----------|
| 1 | Introduction | 1 |
| 2 | Technical Background on Android | 7 |
| 2.1 | Android Software Stack | 9 |
| 2.2 | Android Applications | 11 |
| 2.3 | Android Security Design | 12 |
| 3 | AppGuard | 15 |
| 3.1 | Introduction | 17 |
| 3.1.1 | Contributions | 17 |
| 3.2 | AppGuard | 18 |
| 3.3 | Implementation | 20 |
| 3.3.1 | Policies | 20 |
| 3.3.2 | Inliner | 21 |
| 3.3.3 | Management | 23 |
| 3.3.4 | Challenges | 23 |
| 3.3.5 | Deployment | 24 |
| 3.4 | Experimental Evaluation | 25 |
| 3.4.1 | Performance Evaluation | 25 |
| 3.4.2 | Case Study Evaluation | 27 |
| 3.4.3 | Discussion | 31 |
| 3.5 | Conclusion | 31 |
| 4 | Dynamic Method Hook Injection on Android | 33 |
| 4.1 | Introduction | 35 |
| 4.2 | Background | 36 |
| 4.3 | Implementation | 38 |
| 4.4 | Evaluation | 39 |
| 4.5 | Conclusion | 40 |
| 5 | Boxify | 41 |
| 5.1 | Introduction | 43 |
| 5.2 | Background on Android OS | 45 |
| 5.3 | Requirements Analysis and Existing Solutions | 47 |
| 5.3.1 | Objectives and Threat Model | 47 |
| 5.3.2 | Existing Solutions | 48 |

CONTENTS

| | | |
|----------|-------------------------------|-----------|
| 5.4 | Boxify Architecture | 52 |
| 5.4.1 | Design Overview | 52 |
| 5.4.2 | Target | 54 |
| 5.4.3 | Broker | 58 |
| 5.4.4 | System Integration | 63 |
| 5.5 | Evaluation | 64 |
| 5.5.1 | Performance Impact | 64 |
| 5.5.2 | Runtime Robustness | 65 |
| 5.5.3 | Portability | 65 |
| 5.5.4 | Use-cases | 66 |
| 5.5.5 | Security Discussion | 66 |
| 5.6 | Conclusion | 68 |
| 6 | Related Work | 69 |
| 7 | Conclusion | 77 |

List of Figures

| | | |
|------|---|----|
| 2.1 | BACKGROUND: Android software stack | 10 |
| 3.1 | APPGUARD: Architecture overview | 19 |
| 3.2 | APPGUARD: Host restriction example policy | 20 |
| 3.3 | APPGUARD: HTTPS redirection example policy | 21 |
| 3.4 | APPGUARD: Illustration of call-site monitoring | 22 |
| 4.1 | HOOKING: Rewriting approaches | 37 |
| 4.2 | HOOKING: Library usage example | 39 |
| 5.1 | BOXIFY: Android interactions overview | 46 |
| 5.2 | BOXIFY: OS extension instrumentation points | 48 |
| 5.3 | BOXIFY: Rewriting/Inlining instrumentation points | 49 |
| 5.4 | BOXIFY: Dr. Android and Mr. Hide approach | 50 |
| 5.5 | BOXIFY: Architecture overview | 52 |
| 5.6 | BOXIFY: Target process components | 54 |
| 5.7 | BOXIFY: App loading process | 56 |
| 5.8 | BOXIFY: Broker architecture | 59 |
| 5.9 | BOXIFY: Evaluated Android APIs | 60 |
| 5.10 | BOXIFY: Internal API stability | 61 |

List of Tables

| | | |
|-----|--|----|
| 3.1 | APPGUARD: Inliner evaluation | 26 |
| 3.2 | APPGUARD: Runtime performance evaluation | 26 |
| 4.1 | HOOING: Runtime performance evaluation | 40 |
| 5.1 | BOXIFY: Deployment option comparison | 48 |
| 5.2 | BOXIFY: Middleware benchmarks | 64 |
| 5.3 | BOXIFY: Syscall benchmarks | 64 |
| 5.4 | BOXIFY: Benchmark tools | 65 |
| 5.5 | BOXIFY: Supported Android versions | 65 |

1

Introduction

Smart devices have become part and parcel of everyday life, shaping the way we collect, share, and interact with information today. Pocket-size computers like smartphones and tablets accompany their bearers anywhere and come equipped with a plethora of features. Those include a wide range of embedded sensors, such as a microphone, camera, gyroscope, and GPS, as well as features like instant messaging, contact management, and multimedia collections. Combined with cheap data plans and comprehensive cell coverage, smart devices enable end-users to stay online at all times.

A driving force for the adoption of mobile devices is the ability to enhance their functionality through third-party apps. Initially intended for productivity assistance and information retrieval, including email, calendar, weather and stock market information, public demand, and the availability of developer tools drove rapid expansion into other categories, such as gaming, shopping, navigation, education, entertainment, and health. Apps are conveniently discovered and installed through centralized marketplaces such as Google’s Play Store, with the number of apps on that marketplace alone peaking at 3.6 million in 2018 [80]. Today, apps are no longer confined to handheld devices only, but find increasing use in cars, televisions, and smart home devices.

Despite these advantages, the rapidly increasing prevalence of smart devices has created a vast potential for misuse. The plethora of sensitive information they store about their users, such as private messages, emails, contacts, and photos, makes them an attractive target for attackers. Permanent Internet access and a wide range of sensors allow attackers to turn smart devices into surveillance tools that can potentially track their bearer at all times. A typical attack vector to gain access to these devices is to disguise malware as a seemingly harmless app and to distribute it via application marketplaces, which has been shown to be successful even with app vetting processes in place [96, 97]. But malicious actors are not the only threat to the user’s privacy: Overly-curious apps and advertisement libraries collect sensitive user data, often without the user’s knowledge or consent [33, 43]. For instance, social network apps have been criticized for silently downloading and storing the user’s entire address book to the network’s servers to mine the social graph, even of non-platform users. [76]. This practice is particularly prevalent in apps that are seemingly offered free of charge, because the user unknowingly “pays” by disclosing their private information to the app provider [58].

The evolution of today’s mobile app ecosystem was facilitated by the introduction of operating systems tailored explicitly to smart devices. In 2008, a group of companies led by Google, called the Open Handset Alliance, entered the smartphone market with the open-source software stack Android. As of today, Android has become the most popular operating system for mobile devices, with 2.5 billion monthly active users and an 88% share of the global mobile operating system market [79]. This popularity among end-users paired with the sensitivity of available data makes Android the prime target platform for attackers. Despite this, Android’s set of security and privacy features used to be – and still largely continues to be – quite limited. Its application security model,

based on app isolation and access control for critical system resources, used to leave users with only two options: granting access to all requested resources as a whole at installation time, or not installing the app at all. Up to version 6.0, neither permission revocation after the installation of an app nor dynamic permission assignment was supported. While Android more recently started to support checking permissions on first use – thus making it more contextual and transparent for the end-user – app developers still need to actively opt-in to enable this functionality. This results in a situation where app developers dictate the permissions their apps are granted, with any need for customization on the user side being neglected. Furthermore, the current permission system tends to be impractical for many purposes and can not be easily refined. Ironically, customizations of Android’s security model are impeded by its app isolation paradigm, which effectively prevents third-party applications from enforcing custom security policies.

In their efforts to address these limitations, many researchers have proposed modifications or extensions of the Android software stack. The proposed solutions, however, are rarely adopted by Google or the device vendors, thus forcing users to resort to customized aftermarket firmware if they wish to deploy new security extensions on their devices. This, in turn, poses a technological barrier for average users.

In order to find a solution that provides users with an effective and practical means to protect their privacy, this dissertation focuses on systems for security policy enforcement that can be deployed to ordinary, stock Android devices. In particular, we aim for a solution that can be deployed entirely as an app on stock Android without modifications to the firmware or code of the monitored applications. Our work to establish security policy enforcement on the Android application layer explores inline reference monitoring, application virtualization, and dynamic program instrumentation in Androids Java virtual machine. This research resulted in several peer-reviewed publications [P4, P1, P2, P3], which each contributed to the solution presented in this dissertation.

AppGuard Our first approach to address the shortcomings in Android’s security model leverages inline reference monitors to enforce custom security policies on third-party apps. It extends Android’s permission system to impede overly curious behaviors, supports complex policies, and can mitigate vulnerabilities in apps. It thus is the first solution that provides a practical extension of the current Android permission system as it can be deployed to all Android devices without modification of the firmware or *root* access to the smartphone. Experimental analysis shows that we can remove permissions for overly curious apps as well as defend against several real-world attacks on Android phones with negligible space and runtime overhead.

Despite these advances, there are some limitations to this approach. Instrumentation is performed statically and thus cannot adapt dynamically to user requirements. Furthermore, the approach uses caller-side instrumentation, which can be incomplete in certain circumstances, e.g., if application code is dynamically loaded at runtime. We

address these limitations in our paper on Dynamic Method Hook Injection that presents a technique for dynamic callee-side program instrumentation.

Dynamic Method Hook Injection Rewriting all calls to security-relevant methods requires significant space and time, in particular if this process is performed on a smart device. Our work proposes a novel approach to inline reference monitoring that abstains from caller-site instrumentation even in the case where the monitored method is part of a sealed library. To that end, we divert the control flow towards the security monitor by modifying references to security-relevant methods in the Dalvik Virtual Machine’s internal bytecode representation. This method is similar in spirit to modifying function pointers and effectively allows callee-site rewriting. Our initial empirical evaluation demonstrates that this approach incurs minimal runtime overhead.

Another limitation of APPGUARD is that it provides only insufficient app sandboxing functionality [50] as the reference monitor and the untrusted application share the same process space. Hence, they lack the strong isolation that would ensure tamper-protection and non-bypassability of the reference monitor. Moreover, inlining reference monitors requires modification and hence re-signing of applications, which violates Android’s signature-based same-origin model and puts these solutions into a legal gray area.

Boxify To address these shortcomings, we propose a system based on application virtualization and process-based privilege separation to securely encapsulate untrusted apps in an isolated environment. In contrast to all related work on stock Android, we eliminate the necessity to modify the code of monitored apps, and thereby overcome legal concerns and deployment problems that rewriting-based approaches have been facing. We realize our concept as a regular Android app called BOXIFY that can be deployed without firmware modifications or root privileges.

Outline

The remainder of this dissertation is structured as follows: Relevant technical background information on the Android OS and its security architecture is provided in Chapter 2. We present APPGUARD in Chapter 3, our technique for dynamic method hook injection in Chapter 4, and BOXIFY in Chapter 5. Chapter 6 provides an overview of related work and Chapter 7 concludes the dissertation.

2

Technical Background on Android

The following chapter provides background information on the technical specifics of Android to facilitate an understanding of the ensuing chapters (Chapters 3 to 5). More specific background information related to individual chapters will be provided in the respective chapters.

2.1 Android Software Stack

Android OS is an open-source software stack for mobile devices released in 2008 by a group of companies led by Google, called the Open Handset Alliance. The Android operating system is specifically tailored to handheld devices such as tablets or smartphones and has heavily profited from an ever-growing smartphone market since its launch. Today, Android has become the dominant mobile operating system with a worldwide market share of 88% [79], the current version being 9.0 (called *Pie*). At the time of writing of the last contribution to this thesis, the market share was already about the same, with the latest version then being 5.1.1 (*Lollipop*). Android's success is driven by a wide variety of apps available for the platform. These can be conveniently found, browsed, and downloaded through several centralized market places, the biggest one being Google's Play Store, with the number of apps on this market place alone peaking at 3.6 million in 2018 [80].

The Android software stack (see Figure 2.1) consists of a Linux kernel, a middleware framework comprised of libraries and APIs written in C, and application software running on an application framework, which includes Java-compatible libraries. In the following, we describe these components in more detail.

Linux Kernel The kernel is at the core of every modern operating system, facilitating the communication between hardware and software and providing all essential operating system services, such as memory management, hardware abstraction, and inter-process communication (IPC). The Linux kernel used in Android OS has been slightly modified to fulfill the specific requirements of mobile devices, for example, low-frequency CPUs or scarcity of available resources such as power and memory. For their Android OS, Google uses a custom IPC mechanism initially not part of the Linux kernel called *Binder*, a lightweight implementation of *OpenBinder* developed by Palm Inc., to facilitate cross-process boundary calls.

Runtime and Native Libraries The Android middleware built on top of the Linux kernel consists of native libraries (e.g., Android's *bionic* LibC, OpenGL and SSL), and the application framework. Furthermore, it contains the Android Runtime (ART), which replaced the Dalvik Virtual Machine (DVM) from Android version 5.0 onwards. While the DVM, which executes Dalvik executable (DEX) bytecode, was already optimized for resource-constrained mobile devices, ART utilizes ahead-of-time compilation to further improve runtime performance and efficiency of apps - at the cost of increased binary size.

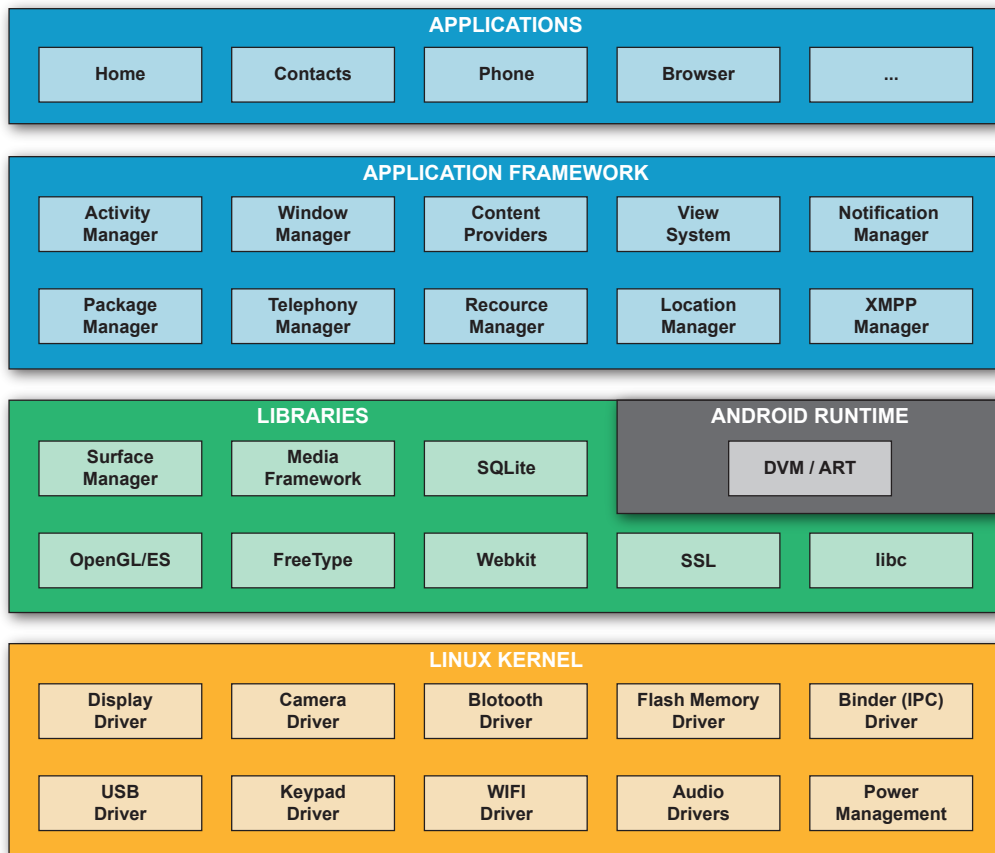


Figure 2.1: Components of the Android software stack.

ART comes equipped with an on-device compiler that translates DEX bytecode into native code specific to and optimized for the concrete hardware platform. To further reduce installation time, a just-in-time compiler was added with Android 7.0.

Application Framework The application software on Android devices is running on an application framework, which is another part of the middleware comprised of core system services written primarily in Java. The application framework implements most of Android’s feature-rich application API that allows app developers to access functionality such as retrieving geolocation or capturing audio from the microphone. This functionality is exposed through several system services (e.g., notification manager, location manager, GUI manager), each of which is responsible for one specific system resource. These system services are built on top of Binder IPC and are remotely callable via well-defined interfaces, which are specified in the Android Interface Definition Language (AIDL). From AIDL, required boilerplate marshaling code for both the receiver and the sender can be generated. Only a few framework services implement the

Binder interface directly without relying on AIDL.

System Apps At the application layer itself, a set of default system apps such as browser, phone, or contacts, completes the application framework. These default system apps are technically identical to third-party apps in as far as they build upon the same framework APIs. However, they come pre-installed with the Android firmware, which enables them to gain privileges to an extent not available to third-party apps. Unlike the application framework, system apps are generally optional, and therefore many device manufacturers often opt to replace them by custom implementations.

2.2 Android Applications

Java used to be the go-to language for developers writing Android applications. More recently, however, Kotlin has begun to see increasing use due to it being a modern, lightweight programming language where developers have to write significantly less code, a development further facilitated by the inherent interoperability between Java and Kotlin. The Android toolchain then compiles the application source to dex bytecode for execution by the Android Runtime (DVM or ART) in use on the device. For performance-critical operations or low-level interactions with the operating system such as direct memory access, Android furthermore supports the use of native code libraries as part of Android applications. Using the Java Native Interface (JNI), application developers can easily invoke functions in these native code libraries from their Java or Kotlin applications.

Android applications consist of four fundamental types of app components, acting as entry points to the app: Activities, Services, Content Providers, and Broadcast Receivers.

- **Activities:** Foreground components displaying a single screen with a user interface.
- **Services:** Components used for long-running background operations such as fetching data from the network or playing music in the background. The functions provided by a service are not exclusive to the application defining that component but can also be made available to other applications on the device. To that end, services expose an interface (typically specified in AIDL) that can be accessed using Binder IPC.
- **BroadcastReceivers:** Components responding to events external to the app, enabling the app to react to system-wide broadcasts even if it's not currently running. While many broadcasts, such as incoming SMS, are sent by the system, apps can also initiate broadcasts, for example, to let other apps know that a download was completed.

- **ContentProviders:** Components exposing structured application data to other apps, through which those apps can query or modify the data if the content provider allows it. The Android system, for example, provides a content provider that manages the user's contact information.

These Android application components establish *inter component communication* (ICC) through Android's Binder IPC mechanism. With Binder as the basic building block for this communication, developers do not interact with Binder directly, but only through Android's ICC abstractions. There are two different layers of abstraction used by Android: the Android Interface Definition Language (AIDL) mentioned earlier and Intents. AIDL enables developers to declaratively specify remotely-callable interfaces, e.g., to create bindable Service components, which is extensively used for the system services in Android's application framework. Unlike AIDL, where a concrete recipient component needs to be known in advance, ICC via Intents defers the resolution of the recipient to the Android system. The app only describes the high-level operation that should be performed, while Android makes sure it is delivered to the right receiver. An example of such an Intent would be capturing an image from the camera, where the concrete component providing the captured image is irrelevant to the initiator of the Intent.

2.3 Android Security Design

With its ascent towards one of the most popular operating systems for mobile devices over the last decade, Android became a prime target for attackers. To counteract these threats, Android developed a security concept based on app isolation and access control for critical system resources.

App isolation is achieved by forcing each application to run in a separate, sandboxed environment that isolates data and code execution from other apps. In contrast to traditional desktop operating systems where applications run with the privileges of the invoking user, Android assigns a unique Linux user ID (UID) to every application at installation time. It is possible for multiple apps from the same developer, identified by their developer signature, to share a UID. Based on this UID, the components of the Android software stack enforce access control rules that govern the application sandboxing.

Furthermore, Android introduces *permissions*, privileges that an app requests at installation time, which the user has to grant to install the app. As of version 6.0, Android supports prompting for user consent once a permission is first required, making permission requests more contextual for the user. Permissions are a crucial component to ensure privilege separation between apps. They are specified by the developer via the XML manifest file that comes with every application package, thus enabling developers to define their app's security policy. Most permissions are String labels that

are immutably assigned to the respective app's UID. These permissions, without which an app cannot access any resources that are security- or privacy-sensitive, are enforced at two different points in the system architecture: the kernel and the middleware.

Every app process, like any other Linux process, uses system calls to the kernel to access low-level resources. The kernel then enforces discretionary access control (DAC) on these system calls based on the UID of the application process to ensure that the process in question has the necessary permissions to issue that call. For instance, each app has a private directory that is not accessible by other applications, which is enforced by the kernel's DAC. Since Android version 4.3, this discretionary access control is complemented with SELinux mandatory access control (MAC) to harden the system against low-level privilege escalation attacks and to reinforce this UID-based compartmentalization.

Furthermore, apps can interact with highly privileged resources via the Android middleware APIs. They are, however, prohibited direct access to these resources, in order not to jeopardize system security and stability. The permission check to determine whether an app has the necessary privileges to call a particular API is performed within the respective system service or app; a centralized policy for checking permissions does not exist. Instead, any framework service providing security- or privacy-critical methods must enforce the corresponding permission connected to the system resources it might expose. This check is facilitated by the Binder IPC mechanism, which provides the calling app's UID to the callee, allowing it to determine whether their current caller possesses the required permissions. If that is not the case, it can throw an exception or take otherwise appropriate action.

3

AppGuard

Enforcing User Requirements on Android Apps

3.1 Introduction

As the development of new security concepts did not keep pace with the emergence of new features, the rapidly growing number of mobile devices has created a vast potential for misuse. Android’s security concept is based on isolation of third-party apps and access control. While this model provides users with the opportunity to review permissions at install time, they are limited to a binary decision: either they grant all permissions - or do not install the app at all. Users can neither dynamically grant and revoke permissions at runtime, nor add restrictions according to their personal needs. Furthermore, users are often not aware of a permission’s impact. They usually do not have enough information to judge whether a permission is indeed required to fulfill a certain task. Research has shown that this is the case even for Android app developers [65, 40].

In order to overcome these limitations, researchers have proposed several approaches. At the time of writing there had been work [64, 63, 62, 23] focused on extending the current permission system, e.g. to prevent critical permission combinations [30], but most approaches [17, 33, 40, 32, 41, 67] targeted the detection of privacy leaks and malicious third-party apps. However, the vast majority relied on modifications of the underlying software stack, which prevented deployment to off-the-shelf Android phones, thus rendering them unsuitable for our objective of putting users back in control of their data.

3.1.1 Contributions

In this work we present a novel policy-based security framework for Android to address the aforementioned limitations of Android’s security model, proposing solutions for a variety of situations, in particular for:

1. *Revoking Android permissions dynamically.* Permissions can be granted and revoked at any time after installation of an app. Graceful revocation is supported by selectively suppressing undesired operations without terminating the program.
2. *Enforcing complex stateful and fine-grained security policies.* Policies can transform the sequence of program actions in case the program deviates from the security policy. Security policies can be arbitrary predicates over that sequence, enabling enforcement of any security property [56].
3. *Policy-based quick-fixes for vulnerabilities in third-party applications.* When an app uses the API in an insecure way, the execution can be transformed into a secure alternative. While it would be cleaner to fix the app directly, a rapid work-around like this provides a temporary solution until the vendor provides an update.

4. *Policy-based mitigation for vulnerabilities in the operating system itself.* Known OS vulnerabilities have always posed a major threat to system security [18]. Our framework defends against malicious apps that try to exploit a vulnerability, which is paramount to relieve the *late update problem*, where vendors integrate the fix at a later time than Google, or do not provide security patches at all.

We built a prototypical implementation called APPGUARD that supports all features listed. Our system does not require any modification to the core software stack of the Android device and thus supports widespread deployment as a stand-alone app. APPGUARD is based on inline reference monitoring [34]. It takes user-defined policies as input and produces a secured self-monitoring app. Our evaluation on typical Android apps has shown very little overhead in terms of space and runtime. The case studies demonstrate the effectiveness of our approach: we successfully limited excessive curiosity of apps, demonstrated complex policies and prevented several real-world attacks on Android phones. At the time of publication, APPGUARD was the first practical defense against these attacks on devices with standard firmware.

3.2 AppGuard

Android’s current permission system fails to address some challenges of mobile applications: apps basically dictate the permissions deemed necessary and users have to grant them as requested short of not installing the application. Therefore, the individual security requirements of users are ignored as they are not really given a choice. In particular, they are unable to grant each permission individually and to grant and revoke certain permissions at a later time. This deficiency becomes particularly severe as not only users have problems with Android’s permission system. Song et al. [65] have shown that even application developers have problems requesting the “correct” permissions for the functionality of their applications, since the Android documentation lacks completeness. If in doubt, they will therefore request too many permissions in order to make their application work.

APPGUARD puts the user back in charge of application permissions. With our system users are given the opportunity to decide both at installation time of an app, as well as at any later point in time what an app is allowed to access. APPGUARD provides more fine-grained policies than Android does itself, and, in contrast to other proposed solutions [62, 23], it is oblivious of the installed Android firmware. It can be installed on all existing Android phones without modifying the core system, which enables wide-spread deployment.

Runtime policy enforcement for third-party applications is not an easy feat on unmodified Android systems. Android’s security concept strictly isolates different applications installed on the same device, preventing them from interfering with each other at runtime. Furthermore, applications cannot gain elevated privileges that would enable them to observe the behavior of other applications. Communication between

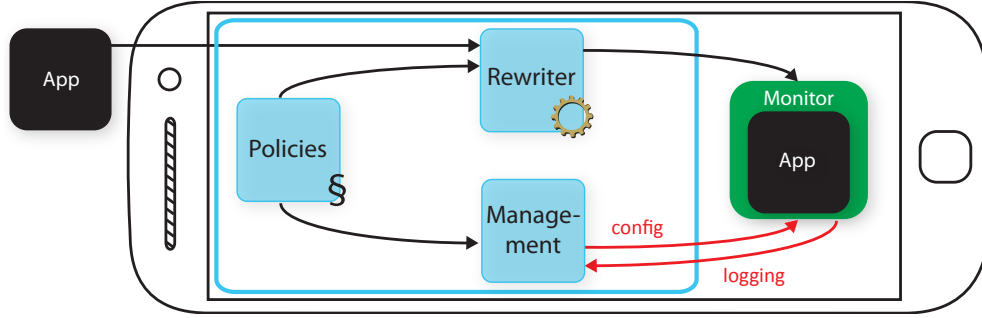


Figure 3.1: High-level architecture of APPGUARD

apps is only possible via Android’s inter-process communication (IPC) mechanism. So far, such communication requires both parties to cooperate, rendering this channel unsuitable for a generic runtime monitor.

APPGUARD tackles this open problem by following an approach pioneered by Erlingsson and Schneier [35] called *inline reference monitor* (IRM). The basic idea is to rewrite an untrusted application such that the code that monitors the application is directly embedded into its code. To this end, IRM systems incorporate a *rewriter* or *inliner* component, that injects additional security checks at critical points into the application bytecode. This enables the monitor to observe a trace of *security-relevant events*, which typically correspond to invocations of trusted system library methods from the untrusted application. To actually enforce a *security policy*, the monitor controls the execution of the application by suppressing or altering calls to security-relevant methods, or even by terminating the program if necessary.

In the IRM context, a policy is typically specified by means of a security automaton that defines which sequences of security-relevant events are acceptable. Such policies have been shown to express exactly the policies enforceable by runtime monitoring [78]. Ligatti et al. differentiate security automata by their ability to enforce policies by manipulating the trace of the program [56]. Some IRM systems [35, 28] implement simple truncation automata, which can only terminate the program if it deviates from the policy. However, this is often undesirable in practice. In their paper [56], Ligatti et al. formulate the notion of *edit automata*, which can transform the program trace by inserting or suppressing events. Monitors based on edit automata are able to react gracefully to policy violations, e.g. by suppressing an undesired method call and returning a dummy value, but allowing the program to continue.

APPGUARD is an IRM system for Android with the transformation capabilities of an edit automaton. Figure 3.1 provides a high-level overview of our system. We distinguish three main components:

1. A set of security policies. APPGUARD provides various Android-specific security policies that govern access to platform API methods that are protected by coarse-grained Android permissions. These methods comprise e.g. methods for reading personal data, creating network sockets, or accessing device hardware like the

```
1 class InternetPolicy extends Policy {  
2   @MapSignatures({ "Ljava/net/URL;->openConnection()" })  
3   public void checkConnection(URL url) throws Exception {  
4     if (!"wetter.com".equals(url.getHost()))  
5       throw new IOException();  
6   }}
```

Figure 3.2: Example policy protecting calls to `java.net.URL.openConnection()`. The callback method `checkConnection(URL)` allows connections to one host only.

GPS or the camera. As a starting point for the security policies, we used the Android permission map by Song et al. [65].

2. The program rewriter. Android applications run within a custom register-based Java VM called *Dalvik*. Our rewriter manipulates Dalvik executable (`dex`) bytecode of untrusted Android applications and generates monitoring code according to the policies to harness the untrusted app.
3. A management component. It offers a graphical user interface that allows the user to set individual policy configurations on a per-application basis. In particular, policies can be turned on or off and be parameterized. In addition, the management component keeps a detailed log of all security-relevant events, enabling the user to monitor the behavior of an application.

3.3 Implementation

APPGUARD is a stand-alone Android application written in pure Java and comprises about 6500 lines of code. It builds upon the *dexlib* library, which is part of the *smali* disassembler for Android by Ben Gruver [45], for manipulating `dex` files. The size of the application package is roughly 750 Kb.

3.3.1 Policies

In our system, a policy is defined by a set of security-relevant method signatures and corresponding callback methods. In our system policies are implemented as Java classes. The callback methods are mapped to a set of method signatures using a custom method annotation `MapSignatures`. Consider Figure 3.2 as a basic example. This policy guards access to the `openConnection()` method in the `java.net.URL` class and only allows connections to the host “wetter.com”.

A policy callback method gains access to the arguments of the original method call by declaring an equivalent list of parameters. In our example, the `checkConnection(URL)` callback method uses the `URL` parameter to decide whether a connection should be allowed. If allowed, the callback method will simply return such that the original

```

1 class HttpsRedirectPolicy extends Policy {
2     @MapSignatures({"Ljava/net/URL;->openConnection()"})
3     public void checkConnection(URL url) throws Exception {
4         if (redirectToHttps(url)) {
5             URL httpsUrl = new URL("https", url.getHost(), url.getFile()
6                                     );
7             URLConnection returnValue = httpsUrl.openConnection();
8             throw new MonitorException(returnValue);
9         }
10    }
11 }

```

Figure 3.3: Example policy that redirects HTTP connections to HTTPS if available.

method call can proceed. If the connection is not allowed, an exception will be thrown that is either caught by the surrounding application code or by a synthetic handler introduced by the inliner. In both cases the original method call will be prevented by this scheme. Details will be discussed in the next subsection.

Furthermore, policies can be stateful and store security state information in member variables of the policy class. The values of these variables are preserved across callback method invocations. Member variables could also be used to store the complete history of intercepted methods.

In general, policy callbacks can perform arbitrary operations. As an example, consider a policy that intercepts HTTP connections and relays them to encrypted HTTPS, if available (see Figure 3.3.) After calling the original method with the new arguments, it throws an exception containing the returned value, which will be substituted for the return value of the original method as described in the next section.

3.3.2 Inliner

The task of the inliner component is to divert the control flow of the target application to the monitoring code at invocation instructions to security-relevant methods. There are two strategies for passing control to the monitor: Either at the call-site in the application code, right before the invocation of the security-relevant method, or at the callee-site, i.e. at the beginning of the security-relevant method. The latter strategy is simpler and more efficient, because callee sites are easier to identify and less in number [7]. Unfortunately, callee-site rewriting is not applicable in our scenario, as security-relevant methods are declared in the trusted Android platform libraries, which are part of the non-modifiable firmware image. Thus, our inliner implements a call-site control flow diversion strategy. The inlining process consists of three steps:

1. Merging the policy classes into the target application's `classes.dex` file, which contains all Dalvik bytecode for the app.
2. Generating the `MonitorInterface`, which serves as a bridge between application code and policies.




| Original snippet | After inlining |
|--|---|
| <pre>1 URL url = new URL(loc); 2 URLConnection conn = 3 url.openConnection();</pre> | <pre>1 URL url = new URL(loc); 2 URLConnection conn; 3 try { 4 MonitorInterface. checkConnection(url); 5 conn = url.openConnection(); 6 } catch (MonitorException e)  { 7 conn = (URLConnection) e. value(); 8 }</pre> |

Figure 3.4: Illustration of the call-site monitoring code for the security-relevant method `java.net.URL.openConnection()`.

3. Injecting monitoring code around invocations of security-relevant methods.

The policy classes are stored precompiled in a separate dex file. In the first step, the inliner copies all class declarations from this file to the `classes.dex` file of the untrusted application.

In order to connect invocations of security-relevant methods to their policy callbacks, the inliner generates a utility class called `MonitorInterface` as the second step of the process. For each security-relevant method specified by the policies, the inliner generates a static guard method in the `MonitorInterface` class. The purpose of this guard method is two-fold: First, callback methods of different policies may be defined for a single security-relevant method. Thus, the guard method invokes all policy callbacks defined for this method signature. Second, the guard method shares the signature of the security-relevant method, including the receiver object for virtual calls, which is passed as the first method argument, if available. Thus, calls to the `MonitorInterface` require only minimal modifications to the application code.

In the final step, the inliner identifies all call sites of security-relevant methods. If a matching instruction is found, the inliner adds monitoring code around the method call as depicted in Figure 3.4. First, a method call to the corresponding guard method in the `MonitorInterface` is inserted right before the invocation of the security-relevant method. Second, the inliner adds a new `try/catch`-block around the inserted guard and the original method call. This block enables policy callbacks to pass a return value to the application code if the original call is suppressed, which also allows to protect security-relevant constructor methods. To this end, policy callbacks can throw a special `MonitorException` that carries the return value to the application code. In the inserted catch block this value is assigned to the intended variable (possibly after type conversion).

As pointed out earlier, another option for the policy is to throw an exception that will be caught by the original application code. Our example policy in Figure 3.2 makes

use of this technique: If a connection is not allowed, the policy callback throws an `IOException`, which resembles the behavior of the original `URL->openConnection()` method if a connection error occurs.

3.3.3 Management

The management component of APPGUARD allows for monitoring the behavior of inlined apps and for configuring policies at runtime. The policy configuration is provided to the inlined app as a world-readable file. Its location is hardcoded into the monitor code during the inlining process. This is motivated by the fact that invocations of security-relevant methods can occur before the inlined application is fully initialized and able to perform Android IPC.

The management component provides a log of all security-relevant events, which enables the user to make informed decisions about the current policy configuration. The log is maintained based on the security-relevant method invocations encountered in the self-monitoring application, which sends its events to the management application. For this direction of the communication we are leveraging a standard Android Service component. The asynchronous nature of Android IPC is not an issue, since security-relevant method invocations that occur before the service connection is established are buffered locally.

3.3.4 Challenges

In our implementation we faced two main difficulties: The handling of reflection and the handling of virtual methods. In the following we will discuss both in detail.

3.3.4.1 Reflection

The Java Reflection API enables the inspection and manipulation of classes, fields, methods, and constructors at runtime without knowing their names at compile time. It can also be used to instantiate new objects and to invoke methods. The latter two features are relevant to our IRM system as they provide alternative ways to invoke security-relevant methods without their signatures appearing in the application bytecode.

We deal with reflection by monitoring critical methods of the Reflection API. To this end, we implement a `ReflectionPolicy` that guards invocations of `java.lang.reflect.Method->invoke()`, `Constructor->newInstance()`, and `java.lang.Class->newInstance()`. Whenever one of these methods is invoked, the policy identifies the target method of the reflective call by inspecting the method parameters. If the target matches a security-relevant method, the policy reflectively invokes the corresponding guard method in the `MonitorInterface` with the arguments of the reflective call. This scheme ensures that the Reflection API cannot be used to circumvent the inlined monitor.

3.3.4.2 Virtual methods

Virtual methods are a core concept of object-oriented programming. The target of a virtual method invocation is not determined statically, but dynamically based on the runtime type of the receiving object. More specifically, an `invoke-virtual` instruction with static target $A \rightarrow m$ can be resolved to any method $B \rightarrow m$ at runtime, where B is a subclass of A or A itself. Thus, virtual method invocations require special treatment by our inliner.

We analyze the class hierarchy of application code before the rewriting step and identify classes that inherit security-relevant methods. If a class does *not* override the security-relevant method, we add the signature of the inherited method to the set of security-relevant methods. The new signature is associated with the same guard method as the security-relevant method in the base class. If the class *does* override a security-relevant method, we can safely ignore invocations referencing the overridden method because any calls to the base class within the overridden method will be already protected by the inliner.

At the current state of the implementation, policy callbacks have to examine the runtime type of the receiver if they want to parameterize according to the target object. In our experiments, we have not encountered any case where this was required.

3.3.5 Deployment

In comparison to existing approaches APPGUARD’s novel security framework can be used on existing Android phones without requiring any changes to the operating system. In particular, it does not require *root* access to the smartphone at any time.

Third party applications installed on Android are usually assigned distinct user ids. By default, application A can neither access nor modify application B .¹ Therefore, our rewriter cannot simply modify already installed applications. Instead, we leverage the fact that installation packages of third party applications can be read from the file system by any application. We read and unpack the application packages of the application to be secured, inline the security monitor, and finally repackage and reinstall the application. In order to start this installation process, the user is asked in a preparatory step to uninstall the existing version of the application. Afterwards, it is possible to install the repackaged application without further problems.

As mentioned, all Android applications need to be signed with a developer key. Since our rewriting process breaks the original signature, we sign the modified app with a new key. However, apps signed with the same key can access each other’s data if they declare so in their manifests. Thus, rewritten apps are signed with keys based on their original signatures in order to preserve the intended behavior. In particular, two apps that were originally signed with the same key, are signed with the same new key after

¹Applications signed with the same developer key and those that have the “shared user id”-flag set constitute special cases.

the rewriting process.

Moreover, we ask the user to enable the OS-option to allow installation of apps that have not been signed by the Google Android market. Due to these two user interactions, no additional root privileges are required for APPGUARD.

3.4 Experimental Evaluation

In this section, we present the results of our experimental evaluation. It focuses on the performance of our framework and the evaluation of its effectiveness in different case studies. As testbed we used the Google Galaxy Nexus smartphone with Android 4.0.2. It has a dual-core 1.2 GHz ARM CPU from Texas Instruments (OMAP 4460) and features 1GB RAM. For our off-the-phone evaluation we use a notebook with an Intel Core i5-2520M CPU (2.5 Ghz, two cores, hyper-threading) and 8GB RAM.

3.4.1 Performance Evaluation

APPGUARD modifies apps installed on an Android device by adding code at the bytecode level. We analyze the time it takes to inline an app and its impact on both size and execution time of the modified app.

Table 3.1 provides an overview of our performance evaluation for the inlining process. We have tested APPGUARD with 13 apps and inlined each of the apps with 9 policies (see Section 3.4.2 for details on the policies). In particular, we list the following results for each of the apps: size of the original application package (Apk), size of the `classes.dex` file before and after the inlining process (Dex and Inl, respectively) and the resulting file size difference (Diff), total number of instructions in the application code (Total), number of instructions that have been instrumented by the inliner (Chg), and, finally, the duration of the whole inlining process, both on the laptop and smartphone (PC and Phone, respectively).

The size of the `classes.dex` file increases on average by approximately 45 Kb. The majority of this increase results from merging the monitoring framework and policy class definitions into the application code, while the inserted security checks only have a minor influence on the file size. The applications in our benchmark exhibit significant differences in the total number of instructions as well as in the size of the application package. These differences are reflected in the execution times of the inliner. In most cases, the total instruction count has the largest impact on the runtime, as all instructions in the application code need to be scanned in order to identify invocations of security-relevant methods. For a few apps (e.g. Angry Birds), however, the runtime is dominated by re-building and compressing the application package file (which is essentially a zip archive). The evaluation also clearly reveals the difference in computing power between the laptop and the phone. While the inlining process takes considerably more time on the phone than on the laptop, we argue that this should not be a major

Table 3.1: Inliner evaluation: sizes of apk file, classes.dex, inlined classes.dex, diff. of dex file, # of total and changed instructions, inlining time on PC and phone.

| App (Version) | Size [Kb] | | | | Instructions | | Time [sec] | |
|-----------------------|-----------|------|------|------|--------------|-----|------------|-------|
| | Apk | Dex | Inl | Diff | Total | Chg | PC | Phone |
| Angry Birds (2.0.2) | 15018 | 994 | 1038 | +44 | 79311 | 100 | 6.5 | 43.4 |
| Barcode Scanner (4.0) | 508 | 352 | 397 | +45 | 46337 | 31 | 1.8 | 4.1 |
| Chess Free (1.55) | 2240 | 517 | 561 | +45 | 52615 | 71 | 3.2 | 7.9 |
| Dropbox (2.1.1) | 3252 | 869 | 913 | +44 | 90334 | 86 | 1.9 | 14.1 |
| Endomondo (7.0.2) | 3263 | 1635 | 1680 | +45 | 134452 | 88 | 2.6 | 23.0 |
| Facebook (1.8.3) | 4013 | 2695 | 2744 | +48 | 224285 | 218 | 3.2 | 47.3 |
| Instagram (1.0.3) | 12901 | 3292 | 3337 | +46 | 254032 | 137 | 4.2 | 66.4 |
| Post mobil (1.3.1) | 858 | 1015 | 1056 | +41 | 84407 | 58 | 1.7 | 11.6 |
| Shazam (3.9.0) | 3904 | 2642 | 2690 | +48 | 259644 | 221 | 2.8 | 47.5 |
| Tiny Flashlight (4.7) | 1287 | 485 | 531 | +46 | 46878 | 109 | 1.8 | 7.3 |
| Twitter (3.0.1) | 2218 | 764 | 813 | +48 | 105594 | 107 | 3.6 | 16.7 |
| Wetter.com (1.3.1) | 4296 | 958 | 1000 | +43 | 89655 | 36 | 2.2 | 15.7 |
| WhatsApp (2.7.3581) | 5155 | 3182 | 3230 | +48 | 437874 | 235 | 3.0 | 57.5 |

Table 3.2: Runtime comparison with micro-benchmarks for function calls in unmodified apps and inlined apps with policies disabled and enabled. The runtime overhead is presented for the inlined app with disabled policies.

| Function Call | Original App | Inlined App | | Overhead |
|--------------------------|--------------|---------------|--------------|----------|
| | | Pol. disabled | Pol. enabled | |
| Socket-><init>() | 0.2879 ms | 0.3022 ms | 0.0248 ms | 5.0% |
| ContentResolver->query() | 10.484 ms | 11.138 ms | 0.1 ms | 6.2% |
| Camera->open() | 150.8 ms | 152.36 ms | 0.6 ms | 1.0% |

concern as the inliner is only run once per application.

The runtime overhead introduced by the inline reference monitor is measured through micro-benchmarks (see Table 5.3.) We compare the execution time of single function calls in three different settings: the original code with no inlining as well as the inlined code with disabled and enabled policies (i.e. policy enforcement turned on or off) . Additionally, we present the overhead incurred for the case where policies are disabled. We list the average execution time for each function call. For the case where we enforce policies we prevent the execution of the respective function.

For all function calls the instrumentation adds a small runtime overhead due to additional code. However, when enabled policies prevent the particular function call, the control flow change leads to a smaller overall execution time. Therefore, it is incomparable to the other execution times, so that we compute the overhead only for the disabled policies. In either case, the incurred runtime overhead is negligible and does not adversely affect the application’s performance.

3.4.2 Case Study Evaluation

The conceptual design of APPGUARD focuses on flexibility and introduces a variety of possibilities to enhance Android’s security features. In this section, we evaluate our framework in several case studies by applying different policies to real world apps from Google’s application market Google Play. As a disclaimer, we would like to point out that we use apps from the market for exemplary purposes only, without implications regarding their security unless we state this explicitly.

For our evaluation, we implemented 9 different policies. Five of them are designed to revoke critical Android platform permissions, in particular the Internet permission (`InternetPolicy`), access to camera and audio hardware (`CameraPolicy`, `AudioPolicy`), and permissions to read contacts and calendar entries (`ContactsPolicy`, `CalendarPolicy`). Furthermore, we introduce a complex policy that tracks possible fees incurred by untrusted applications (`CostPolicy`). The `HttpsRedirectPolicy` and `MediaStorePolicy` address security issues in third-party apps and the OS. Finally, the `ReflectionPolicy` described in Section 3.3.4.1 monitors invocations of Java’s Reflection API. In the following case studies, we highlight 6 of these policies and evaluate them in detail on real-world apps.

Our case studies focus on (a) the possibility to revoke standard Android permissions - which is arguably the feature Android users desire most. Additionally, it is possible to (b) enforce fine-grained permissions that are not supported by Android’s existing permission system, and, (c) to enforce complex and stateful policies based on the current execution trace. Finally, our framework provides quick-fixes and mitigation for vulnerabilities both in (d) third-party apps and (e) the operating system.

(a) Revoking Android permissions

Many Android applications request more permissions than necessary for achieving the intended functionality. A prominent example is the Internet permission *android.permission.INTERNET*, which allows sending and receiving arbitrary data to and from the Internet. Although the majority of applications requests this permission, it is not required for the core functionality of an app in many cases. It is often used just for providing in-app advertisements. However, overly curious apps that, e.g., upload the user’s entire contact list to their servers, and even trojan horses are reported on a regular basis. Unfortunately, users cannot simply add, revoke, or configure permissions dynamically at a fine-grained level. Instead, users have to decide at installation time whether they accept the installation of the app with the listed permissions or they reject them with the consequence that the app cannot be installed at all.

APPGUARD overcomes this unsatisfactory all-or-nothing situation by giving users the chance to safely revoke permissions at any time at a fine-grained level. We aim at a “safe” revocation of permissions, so that applications with revoked permissions will not be terminated by a runtime exception. To this end, we carefully provide proper dummy return values instead of just blocking unsafe function calls [52]. We tested the

revocation of permissions on several apps, of which we highlight two in the following.

Case study: Twitter

As an example for the revocation of permissions, we chose the official app of the popular micro-blogging service Twitter. It attracted attention in the media [76] for secretly uploading phone numbers and email addresses stored in the user's address book to the Twitter servers. While the app requested the permissions to access both Internet and the user's contact data, it did not indicate that this data would be copied off the phone. As a result of the public disclosure, the current version of the app now explicitly informs the user before uploading any personal information.

We can stop the Twitter app from leaking any private information by completely blocking access to the user's contact list. The contact data is used as part of Twitter's "Find friends" feature that makes friend suggestions to new users based on information from their address book. Since friends can also be added manually, APPGUARD leverages the `ContactsPolicy` to protect the user's privacy at the cost of losing only minor convenience functionality. Actual policy enforcement is done by monitoring queries to the `ContentResolver`, which serves as a centralized access point to Android's various databases. Data is identified by a URI, which we examine to selectively block queries to the contact list by returning a mock result object. Our tests were carried out on an older version of the Twitter app, which was released prior to their fix.

Case study: Tiny Flashlight

The core functionality of the Tiny Flashlight app is to provide a flashlight, either using the camera's LED flash, or by turning the whole screen white. At installation time, the app requests the permissions to access the Internet and the camera. Manual analysis indicates that the Internet permission is only required to display online advertisements. However, in combination with the camera permission this could in principle be abused for spying purposes, which would be hard to detect without further detailed code or traffic analysis. APPGUARD can block the Internet access of the app with the `InternetPolicy` (see Section 3.3.1 and Figure 3.2), which, in this particular case, has the effect of an ad-blocker. We monitor constructor calls of the various `Socket` classes, the `java.net.url.openConnection()` method as well as several other network I/O functions, and throw an `IOException` if access to the Internet is forbidden.

Apart from the Internet permission, users might not easily see why the camera permission is required for this app. Here, our analysis indicates that – depending on the actual smartphone hardware – the flashlight can in some cases be accessed directly, while in others only via the camera interface. Although requesting this permission seems to be benign for this app, our approach offers the possibility to revoke camera access. We enforce the `CameraPolicy` by monitoring the `android.hardware.Camera.open()` method. The policy simulates hardware without a camera by returning a null value. The Tiny Flashlight app gracefully handles the revocation of the camera permission by falling back to the screen-based flashlight solution.

(b) Enforcing fine-grained permissions

Besides the revocation of existing permissions, it is also possible to design fine-grained permissions that restrict the access of third-party apps. These permissions can add new restrictions to a functionality that is not yet limited by the current permission system and to a functionality that is already protected, but not in the desired way. Here, again, the Internet permission is a good example. From the user’s point of view, most apps should only communicate with a limited set of servers.

The `wetter.com` app provides weather information and should only communicate with its servers to query weather information. The `InternetPolicy` of APPGUARD provides fine grained Internet access enabling a consequent white-listing of web servers on a per-app basis. For this particular app we restrict the Internet access as illustrated in the first case study and extend it with regular-expression-based white-listing: `^(.+\\.com)?wetter\\.com$`. Similar to the Tiny Flashlight app, no more advertisements are shown while the application’s core functionality is preserved. The refined Internet policy can also be applied in a general setting as the white-listing can be configured in the management interface by choosing from a list of hosts the app has tried to connect to in the past.

(c) Enforcing complex and stateful policies

Stateless permissions, as discussed in the previous case studies, cannot be used to enforce policies that depend on the trace of the current execution. Using APPGUARD it is also possible to implement complex stateful policies, e.g. to limit the number of text messages or phone calls to costly numbers, or to block the Internet access after sensitive information like contacts or calendar entries has been accessed.

The `Post mobil` app provided by the German postal service Deutsche Post offers, besides informative services, the possibility to buy stamps online via premium service calls or text messages. To limit the cost incurred by this application, it is necessary to track the number of previous calls. APPGUARD tracks these numbers and provides the `CostPolicy` that limits the number of possible charges. We monitor the relevant function calls for sending text messages and for making phone calls, e.g. `android.telephony.SmsManager.sendMessage()`. In order to monitor the start of phone calls, it is necessary to track so-called *Intents*, Android’s message format for inter- and intra-app communication. *Intents* contain two parts, an *action* to be performed and parameter data encoded as URI. For example, intents that start phone calls have the action `ACTION_CALL`. We track intents by monitoring intent dispatch methods like `android.app.Activity.startActivity(Intent)`.

(d) Quick-fixes for vulnerabilities in third-party apps

Our system can also fix vulnerabilities in third-party applications. As an example, some applications still transmit sensitive information over the Internet via the HTTP protocol. Although most apps use encrypted HTTPS for the login procedures to web servers, there are still some applications that return to unencrypted HTTP after successful login, thereby transmitting their authentication tokens in plain text over the Internet. Attackers could eavesdrop on the connection to impersonate the current user.

The Endomondo Sports Tracker uses the HTTPS protocol for the login procedure only, and returns to the HTTP protocol afterwards, which transmits the unencrypted authentication token. As the Web server supports HTTPS for the whole session, the `HttpsRedirectPolicy` of APPGUARD enforces its usage throughout the session (see Figure 3.3), which protects the user’s account and data from identity theft. Whenever the app attempts to open a HTTP connection, we instead open an HTTPS connection (see the monitored method invocations in the first case study). Depending on the monitored function, we either return the redirected HTTPS connection, or the content from the redirected connection.

(e) Mitigation for operating system vulnerabilities

We also found our tool useful to mitigate operating system vulnerabilities. As we cannot change the operating system itself, we instrument all applications with a global security policy to prevent exploits.

Case study: Access to photos without a permission

An example for an operating system vulnerability is the lack of protection of the user’s photos on Android phones. Any application can access these photos on the phone without any permission check [18]. Together with the Internet permission, an app could copy all photos to arbitrary servers on the Internet. This was demonstrated by a proof-of-concept exploit that – disguised as an inconspicuous timer app – uploads the user’s personal photos to a public photo sharing site.

Android stores photos in a central media store, that can be accessed via the `ContentResolver` object, similar to contact data in the first case study. Leveraging the `MediaStorePolicy`, we block access to the stored photos, successfully preventing the exploit.

Case study: Local cross-site scripting attack

Similar to the mitigation of the photo access bug, it is also possible to fix security vulnerabilities in core applications that cannot be inlined directly. The Android browser that comes with all devices is vulnerable to a local cross-site scripting attack [4] up to Android version 2.3.4. If the Android browser receives `VIEW` intents from another app with an HTTP or HTTPS URI, it opens a new browser window and loads the requested web site. Similarly, it also handles `VIEW` intents with a `javascript:` URI, however, up to Android version 2.3.4, the browser reuses the currently active window. Consequently, the JavaScript code given in the intent will be executed in the context of the current web site, which leads to a local cross-site scripting vulnerability.

This attack can be mitigated by disallowing this combination of intents. The `InternetPolicy` monitors `startActivity(Intent)` calls and throws an exception if the particular intent is not allowed. The same approach can be leveraged to preclude third-party apps with no Internet permission from using intents with an HTTP URI to send data to arbitrary servers on the Internet.

3.4.3 Discussion

The presented framework solves a pressing security problem of the Android platform. Coarse-grained and static policies like the access control mechanism of Android open the door for silent privacy violations and trojan horses, as the user never sees what an application actually does with the requested permissions. Our fine-grained dynamic policies can, e.g., be used to distrust the app and only grant a permission once the user finds that the app does not perform as expected. The logging-based approach in our tool allows a user to see which API calls were denied, possibly with the value of significant parameters. Granting access to those calls that are deemed necessary with restrictions on parameters (like accessible host names) will eventually lead to a minimal set of permissions that fulfills the privacy and security needs of a user.

We demonstrated that our solution is practical, as the runtime overhead and the increase in package sizes are negligible. The actual runtime overhead obviously depends on the complexity of the policy. However, when a policy denies access, the program will in general take a different execution path that usually leads to shorter times. The user experience does not suffer from rewriting the application. In particular, we did not notice any delays using the rewritten app. The rewriting process proceeds fast even on the limited hardware of a mobile phone. The inlining time is already reasonable, but we still see a large potential for reducing this time with some optimizations.

We outline some challenges and future work in the following: We currently do not monitor any code outside of the `classes.dex` file, in particular we might miss code in native libraries accessed via Java's Native Interface (JNI), dynamically loaded classes (from external sources), and external programs accessed via inter-procedure calls.

Android programs are multi-threaded by default. Issues of thread safety could therefore arise in the monitor. While we do not yet offer policies that take the relative timing of method calls in different threads into account, we plan to extend our system to support *race-free policies* [24] in the future.

3.5 Conclusion

We have presented a practical approach to overcome Android's limitations regarding secure, user-driven permission management. The system is based on an inline reference monitor and can be deployed to all Android devices as it does not rely on modifying the firmware. Most prominently, the system can curb the pervasive overly curious behavior of Android apps. Apart from that, we are able to enforce complex stateful security policies and mitigate vulnerabilities of both third-party apps as well as the OS. Our experimental analysis demonstrates that the overhead of both space and runtime are negligible. Further, the case studies illustrate the prevention of several real-world attacks on Android vulnerabilities.

4

Dynamic Method Hook Injection on Android

4.1 Introduction

With APPGUARD, we have introduced a practical approach for security policy enforcement on stock Android devices by leveraging inline reference monitors. To that end, untrusted apps are rewritten to invoke a security monitor before each security-sensitive operation, which is typically a call to a method defined in Android’s system libraries. The monitor checks whether the security policy allows the attempted operation: In the positive case, it lets the original call proceed, while a negative decision blocks the security-sensitive operation. In the latter case, it returns a mock value to prevent the app’s termination due to an exception, if necessary. This variant of inline reference monitoring is called *caller-site rewriting*, as all call sites to security-sensitive operations must be instrumented.

However, this approach has a number of drawbacks. First, it only monitors calls that can be found by static analysis. In particular, this includes calls executed via Java’s Native Interface (JNI) or from dynamically loaded code (e.g., code downloaded post-installation). In consequence, this can lead to incomplete enforcement of the security policy as security-relevant method invocations might bypass the security monitor. Furthermore, the approach requires modifications to the bytecode of the monitored app at *all invocation points* of security-sensitive methods, which requires significant time, in particular, if this process is performed on a smart device.

These issues are addressed by an alternative reference monitor style, called *callee-site rewriting*. This style is far less invasive, as it only instruments the entry of the security-sensitive method itself instead of all the invocation points. On top of that, it also monitors method invocations that are not statically determinable, such as calls executed via Java’s Reflection API or the Java Native Interface mentioned above. Unfortunately, static callee-site rewriting is not feasible for almost all security-relevant code, as this code is defined in sealed libraries (i.e. which cannot be modified) and loaded before any client code executes. Thus static rewriting of these libraries is impossible, rendering *callee-site rewriting* unsuitable for use on Android devices.

Our contribution is that we enable callee-site rewriting for sealed libraries. We achieve this by diverting control flow in the virtual machine. This insight is based on the observation that the VM-internal data structures that represent the libraries in memory are modifiable. Therefore, it is possible to alter the control flow by modifying the reference to the library method’s bytecode, which reroutes a call to this method to another piece of bytecode. For the purpose of inline reference monitoring, we relay an invocation of a security-relevant method to a method that checks whether the security policy allows the original invocation. If this is *not* the case, we simply return (a mock value); otherwise we invoke the original method with the original parameters. As we are altering references to Java bytecode, we have access to all the parameters of the original method call when checking the security policy, just like the caller-site instrumentation does. At the same time, if an app invokes the security-relevant method from JNI or

via reflection, we will still monitor this call as it jumps into the monitor, as well. We are not aware of any previous work that modified references to bytecode in a virtual machine in order to divert control flow to a different functionality.

In more detail, we make the following contributions:

1. We propose to rewrite references to the bytecode of security-relevant methods in-memory in order to achieve the same effects as callee-site rewriting does. To that end, we invoke a native method at program entry that diverts control flow from security-sensitive methods to our monitor methods. We achieve this by modifying the reference to the bytecode of the security-sensitive method and storing the original reference inside the monitor, which effectively makes the original method available to our monitor only.
2. Our in-memory callee-site rewriting only requires minimal instrumentation of an app that needs to be protected by a security policy. In practice, all entry points to the program need to ensure that the references have already been altered. Otherwise, a piece of native code is invoked that modifies these references.
3. In-memory rerouting of security-sensitive methods allows dynamic policy updates and is more efficient than static program rewriting, as it only alters the references of methods that the policy currently protects. Static rewriting would either need to instrument all potentially security-relevant methods or to re-instrument whenever the policy is modified. On top of that, our technique is less invasive, which facilitates on-the-phone instrumentation and minimizes possible conflicts with the original application.
4. We demonstrate the feasibility of our proposed technique by a prototypical implementation. Initial micro-benchmarks show that the dynamic overhead of this technique is minimal and negligible in a practical application.

4.2 Background

Runtime policy enforcement for third-party applications cannot be easily integrated into unmodified Android systems. Android’s security concept strictly isolates different applications installed on the same device to prevent apps from interfering with each other at runtime. Furthermore, applications cannot gain elevated privileges to observe the behavior of other applications. Communication between apps is only possible via Android’s inter-process communication (IPC) mechanism. However, such communication requires both parties to cooperate, rendering this channel unsuitable for a generic runtime monitor.

Several approaches tackled this problem by following an approach pioneered by Erlingsson and Schneider [35] called *inline reference monitor* (IRM). The basic idea is to rewrite an untrusted application such that the code that monitors the application

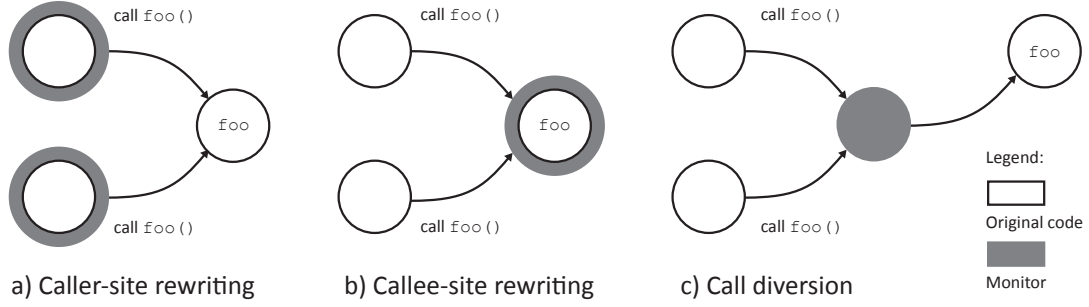


Figure 4.1: Visualization of different rewriting approaches

is directly embedded into its code. To this end, IRM systems incorporate a *rewriter* or *inliner* component that injects additional security checks, called guards, at critical points into the application bytecode. A guard can be injected into the control flow at different positions, but clearly, such a guard should be executed before the critical functionality is executed. There are two semantically equivalent approaches to IRM, as presented in Figure 4.1: *Caller-site-rewriting* (a) adds the guard before every critical call, while *callee-site rewriting* (b) injects the guard into the entry of the critical function itself. The latter is usually more efficient, since the guard only needs to be injected once. Unfortunately, Android’s system libraries are sealed so that inlining the guards into a library is impossible. In order to achieve the same effect as traditional callee-site rewriting we divert all function calls from the security-critical library method to our security guard (see Figure 4.1(c)). Once the guard allows the execution, the original library function is invoked. This redirection, however, incurs an additional method call.

The injected security guards can now efficiently enforce a *security policy*. To actually enforce a policy, the monitor may suppress or alter calls to security-relevant functionality, or even terminate the program if necessary.

In the IRM context, a policy is typically specified by means of a security automaton that defines which sequences of security-relevant events are acceptable. Such policies have been shown to express exactly the policies enforceable by runtime monitoring [78]. Ligatti et al. differentiate security automata by their ability to enforce policies as they manipulate the trace of the program [56]. Some IRM systems [35, 28] implement *truncation automata*, which can only terminate the program if it deviates from the policy. However, this is often undesirable in practice. Ligatti et al. [56] formulate the notion of *edit automata*, which can transform the program trace by inserting or suppressing events. Monitors based on edit automata are able to react gracefully to policy violations, e.g. by suppressing an undesired method call and returning a mock value, while allowing the program execution to continue.

On top of providing an elegant security policy enforcement mechanism, a key aspect of IRM-based security solutions is ease of deployment. User’s need to be able to install the security system without requiring expert knowledge (e.g. gaining root access or changing the smartphone firmware). Our prior work AppGuard [P1, P2] demonstrates

that rewriting apps directly on the phone and subsequently installing the instrumented apps is possible without modifying the base operating system or requiring root access.

4.3 Implementation

Our approach is based on diverting function calls to system libraries to functions in our own library that first perform a security check. The diversion is achieved by replacing the reference to a method's bytecode in the VM's internal representation (e.g. a virtual method table) with the reference to our security guard. Our security guards reside in an external library which is dynamically loaded on application startup. Therefore, we do not need to reinstrument the app when the security policy is modified. Additionally, we store the original reference in order to access the original function later on, e.g., in case the security check grants the permission to execute the security-critical method. In order to ensure instrumentation of security-sensitive methods before their execution, we create an application class that becomes the superclass of the existing application class¹. Our new class contains a static initializer, which is the very first code executed upon application startup. The initializer loads our native C-library using `System.loadLibrary()`.

Invocations of security-critical methods do *not* need to be rewritten statically. Instead, we use Java Native Interface (JNI) calls at runtime to replace the references to each of the functions to be monitored. More precisely, we call the JNI method `GetMethodID()` which takes a method's signature, and returns a pointer to the internal data structure describing that method. This data structure contains a reference to the bytecode instructions associated with the method, as well as metadata such as the method's argument types or the number of registers. In order to redirect the control flow to our guard method, we overwrite the reference to the instructions such that it points to the instructions of the security guard's method instead. Additionally, we adjust the intercepted method's metadata to be compatible with the guard method's code. In particular, we adjust the number of registers to the number of the guard method's registers. This approach works for pure Java methods as well as methods with a native implementation.

We illustrate how to replace a method using the functionality provided by our instrumentation library in Figure 4.2. Calling `Instrumentation.replaceMethod()` replaces the instruction reference of method `foo()` of class `com.test.A` with the reference to the instructions of method `bar()` of class `com.test.B`. It returns the original reference, which we store in a variable `A_foo`. Therefore, subsequently calling `A.foo()` will invoke `B.bar()` instead. The original method can still be invoked by `Instrumentation.callOriginalMethod(A_foo)`. Note that the handle `A_foo` will be a secret of the security policy in practice, therefore the original method can not be invoked directly by the instrumented app.

¹In case no application class exists, we register our class as the application class.

```

1 public class Main {
2     public static void main(String[] args) {
3         A.foo(); // calls A.foo()
4
5         MethodHandle A_foo = Instrumentation.replaceMethod(
6             "Lcom/test/A;->foo()", "Lcom/test/B;->bar()");
7
8         A.foo(); // calls B.bar()
9
10        Instrumentation.callOriginalMethod(A_foo); // calls A.foo()
11    }
12 }

```

Figure 4.2: Example illustrating the functionality of the instrumentation library

Our approach relies only on the layout of Dalvik’s internal data structure for methods, which has not changed since the initial version of Android. However, our instrumentation system could be easily adapted if the layout were to change in future versions of Android.

We are not aware of any possibility to bypass or disable our instrumentation in Java code, as this code is strongly typed. It can even handle cases like reflection or externally loaded libraries, which have not been instrumented. However, native code could potentially alter the references we modified, but it wouldn’t know the original references, as our native code executes first. Native code could also modify the guard’s bytecode instructions or data structures, which is out of the scope of our approach.

4.4 Evaluation

In the following we present the results of our experimental evaluation. We measure the performance overhead of our call diversion approach through several micro-benchmarks (see Table 5.3.) All benchmarks have been executed on a Google Galaxy Nexus smartphone running Android version 4.1.1 (Jelly Bean). The smartphone has a dual-core 1.2 GHz ARM CPU from Texas Instruments (OMAP 4460) and 1GB of RAM. Our techniques require *no* custom firmware, which allows widespread deployment. We envision a instrumentation process similar to our previous work [T1], where a third-party app can be rewritten directly on the phone. The rewriting process only adds code to load the policy classes and executes native code that modifies the references to methods that need to be monitored.

For the evaluation of the runtime overhead we chose to conduct time measurements on three method calls with different runtime complexity, namely `Socket-><init>()`, `ContentResolver->query()`, and `Camera->open()`. We measured time using the `System->nanoTime()` function. One measurement cycle consists of x iterations over the particular function call inside a loop, where $x = 25$ for `Camera->open()`, $x = 500$ for `ContentResolver->query()`, and $x = 10000$ for `Socket-><init>()`.

Table 4.1: Runtime comparison with micro-benchmarks for normal function calls and guarded function calls with policies disabled as well as the introduced runtime overhead.

| Function Call | Original Call | Guarded Call | Overhead |
|--------------------------|---------------|--------------|----------|
| Socket-><init>() | 0.0186ms | 0.0212 ms | 21.4% |
| ContentResolver->query() | 19.5229 ms | 19.4987 ms | 0.8% |
| Camera->open() | 74.498 ms | 79.476 ms | 6.4% |

We executed each cycle 10 times per benchmark. Table 4.1 reports the median runtime for the original function, for the rewritten function with disabled policies (i.e., we directly call the original function), and, finally, the runtime overhead in percent. We do not report the overhead with enabled policies as this would result in negative overhead as the original methods would not be executed.

During the evaluation we found that in a few cases the monitored calls were faster than the original calls, even though we explicitly invoke garbage collection before each cycle to minimize its distortion. These cases are clearly outliers, possibly due to the operating system’s scheduling strategies for other apps running on the same phone. The reported median overhead abstracts from such effects and is always positive. While the relative overhead may seem high, the absolute value is almost negligible and does not adversely affect the application’s performance, in particular as any realistic program only invokes a limited number of security-sensitive methods. The micro-benchmarks give a worst-case approximation of the overhead incurred by a program that would only invoke protected functionality.

4.5 Conclusion

We presented an efficient new approach to inline reference monitoring for Android apps. Our call diversion approach follows the idea of callee-site rewriting and heavily reduces the number of changes that have to be performed during app instrumentation. Furthermore, it reduces the runtime of the inlining process and facilitates on-the-phone instrumentation. Our approach allows for dynamic updates of security policies as only references to bytecode need to be changed. We demonstrated the feasibility of the approach through an experimental evaluation and have integrated it into our prior work APPGUARD.

5

Boxify

Full-fledged App Sandboxing for Stock Android

5.1 Introduction

Security research has shown that the privacy of smartphone users – and in particular of Android OS users, due to Android’s popularity and open-source mindset – is jeopardized by a number of different threats. Those include increasingly sophisticated malware and spyware [96, 97], overly curious libraries [33, 43], but also developer negligence and absence of fail-safe defaults in the Android SDK [44, 37]. To remedy this situation, the development of new ways to protect the end-users’ privacy has been an active topic of Android security research during the last years.

From a deployment perspective, the proposed solutions followed two major directions: The majority of the solutions [30, 62, 64, 12, 23, 98, 74, 83] extended the UID-centered security architecture of Android. In contrast, a number of solutions [54, 88, 27, 70, 26, P1] promote inlined reference monitoring (IRM) [34] as an alternative approach that integrates security policy enforcement directly into Android’s application layer, i.e., the apps’ code. APPGUARD as presented in this dissertation (see Chapter 3) stands in line with this latter research direction.

However, this dichotomy is unsatisfactory for end-users: While OS security extensions provide stronger security guarantees and are preferable in the long run, they require extensive modifications to the operating system and Android application framework. In contrast, solutions that rely on inlined reference monitoring such as APPGUARD avoid this deployment problem by moving the reference monitor to the application layer and allowing users to install security extensions in the form of apps. However, the currently available solutions provide only insufficient app sandboxing functionality [50] as the reference monitor and the untrusted application share the same process space. Hence, they lack the strong isolation that would ensure tamper-protection and non-bypassability of the reference monitor. Moreover, inlining reference monitors requires modification and hence re-signing of applications, which violates Android’s signature-based same-origin model and puts these solutions into a legal gray area.

To overcome this dichotomy we aim for the sweet spot between the two directions that could provide immediate strong privacy protection against rogue applications. It should combine the security guarantees of OS security extensions with the deployability of IRM solutions, while simultaneously avoiding their respective drawbacks. Effectively, such a solution would provide an OS-isolated reference monitor that can be deployed entirely as an app on stock Android without modifications to the firmware or code of the monitored applications. In this chapter we present a novel concept for Android app sandboxing based on *app virtualization*, which provides tamper-protected reference monitoring without firmware alterations, root privileges or modifications of apps.

The key idea of our approach is to encapsulate untrusted apps in a restricted execution environment within the context of another, trusted sandbox application. To establish a restricted execution environment, we leverage Android’s “*isolated process*” feature, which allows apps to totally de-privilege selected components—a feature that

has so far received little attention beyond the web browser. By loading untrusted apps into a de-privileged, isolated process, we shift the problem of sandboxing the untrusted apps from *revoking* their privileges to *granting* their I/O operations whenever the policy explicitly allows them. The I/O operations in question are syscalls (to access the file system, network sockets, bluetooth, and other low-level resources) and the Binder IPC kernel module (to access the application framework). We introduce a novel app virtualization environment that proxies all syscall and Binder channels of isolated apps. By intercepting any interaction between the app and the system (i.e., kernel and app framework), our solution is able to enforce established and new privacy-protecting policies. Additionally, it is carefully crafted to be transparent to the encapsulated app in order to keep the app agnostic about the sandbox and retain compatibility to the regular Android execution environment. By executing the untrusted code as a de-privileged process with a UID that differs from the sandbox app’s UID, the kernel securely and automatically isolates at process-level the reference monitor implemented by the sandbox app from the untrusted processes. Technically, we build on techniques that were found successful in related work (e.g., libc hooking [88]) while introducing new techniques such as Binder IPC redirection through ServiceManager hooking. We realize our concept as a regular app called BOXIFY that can be deployed on stock Android. To the best of our knowledge, BOXIFY is the first solution to introduce *application virtualization* to stock Android.

In summary, we make the following contributions:

1. We present a novel concept for application virtualization on Android that leverages the security provided by *isolated processes* to securely encapsulate untrusted apps in a completely de-privileged execution environment within the context of a regular Android app. To retain compatibility of isolated apps with the standard Android app runtime, we solved the key technical challenge of designing and implementing an efficient app virtualization layer.
2. We realize our concept as an app called BOXIFY, which is the first solution that ports app virtualization to the Android OS. BOXIFY is deployable as a regular app on stock Android (no firmware modification and no root privileges required) and avoids the need to modify sandboxed apps.
3. We systematically evaluate the efficacy and efficiency of BOXIFY from different angles including its security guarantees, different use-cases, performance penalty, and Android API version dependence across multiple Android OS versions.

The remainder of this chapter is structured as follows. In Section 5.2, we provide necessary technical background information on Android. We define our objectives and discuss related work in Section 5.3. In Section 5.4, we present our BOXIFY design and implementation, which we evaluate in Section 5.5. We conclude the paper in Section 5.6.

5.2 Background on Android OS

Android OS is an open-source software stack (see Figure 5.1) for mobile devices consisting of a Linux kernel, the Android application framework, and system apps. The application framework together with the pre-installed system apps implement the Android application API. The software stack can be extended with third-party apps, e.g., from Google Play.

Android Security Model. On Android, each application runs in a separate, simple sandboxed environment that isolates data and code execution from other apps. In contrast to traditional desktop operating systems where applications run with the privileges of the invoking user, Android assigns a unique Linux user ID (UID) to every application at installation time. Based on this UID, the components of the Android software stack enforce access control rules that govern the app sandboxing. To understand the placement of the enforcement points, one has to consider how an app can interact with other apps (and processes) in the system:

Like any other Linux process, an app process uses syscalls to the Linux kernel to access low-level resources, such as files. The kernel enforces discretionary access control (DAC) on such syscalls based on the UID of the application process. For instance, each application has a private directory that is not accessible by other applications and DAC ensures that applications cannot access other apps' private directories. Since Android version 4.3 this discretionary access control is complemented with SELinux mandatory access control (MAC) to harden the system against low-level privilege escalation attacks and to reinforce this UID-based compartmentalization.

The primary channel for inter-application communication is Binder *Inter-Process Communication* (IPC). It is the fundamental building block for a number of more abstract inter-app communication protocols, most importantly *Inter-Component Communication* (ICC) [31] among apps and the application framework. For sandboxing applications at the ICC level, each application UID is associated with a set of platform permissions, which are checked at runtime by reference monitors in the system services and system apps that constitute the app framework (e.g. `LocationService`). These reference monitors rely on the Binder kernel module to provide the UID of IPC senders to the IPC receivers.

In general, both enforcement points are implemented *callee-sided* in the framework and kernel, and hence agnostic to the exact call-site within the app process. This means that enforcement applies equally to all code executing in a process under the app's UID, i.e., to both Java and native code.

Additionally, Android verifies the integrity of application packages during installation based on their developer signature. The corresponding developer certificate is afterwards used to enforce a same-origin policy for application updates, i.e., newer app versions must be signed with the same signing key as the already installed application.

Isolated Process. The *Isolated Process*, introduced in Android version 4.1, is a security

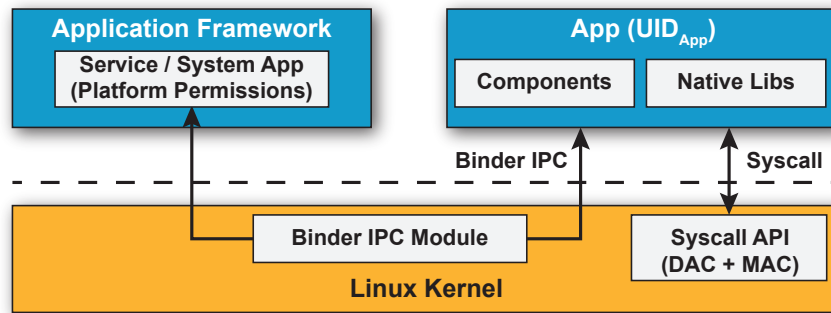


Figure 5.1: High-level view of interaction between apps, application framework, and Linux kernel on Android.

feature that has received little attention so far. It allows an app developer to request that certain service components within her app should run in a special process that is isolated from the rest of the system and has no permissions of its own [2]. The isolated process mechanism follows the concept of *privilege separation* [69], which allows parts of an application to run at different levels of privilege. It is intended to provide an additional layer of protection around code that processes content from untrusted sources and is likely to have security holes. Currently, this feature is primarily geared towards web browsers [47] and is most prominently used in the Chrome browser to contain the impact of bugs in the complex rendering code.

An isolated process has far fewer privileges than a regular app process. An isolated process runs under a separate Linux user ID that is randomly assigned on process startup and differs from any existing UID. Consequently, the isolated process has no access to the private app directory of the application. More precisely, the process' filesystem interaction is limited to reading/writing world readable/writable files. Moreover, the isolated process' access to the Android middleware is severely restricted. The isolated process runs with no permissions, regardless of the permissions declared in the manifest of the application. More importantly, the isolated process is forbidden to perform any of the core Android IPC functions: Sending Intents, starting Activities, binding to Services or accessing Content Providers. Only the core middleware services that are essential to running the service component are accessible to the isolated process. This effectively bars the process from any communication with other apps. The only way to interact with the isolated process from other application components is through the Service API (binding and starting). Further, the transient UID of an isolated process does not belong to any privileged system groups and the kernel prevents the process from using low-level device features such as network communication, bluetooth or external storage. As of Android v4.3, SELinux reinforces this isolation through a dedicated process type. With all these restrictions in place, code running in an isolated process has only minimal access to the system, making it the most restrictive runtime environment Android has to offer.

5.3 Requirements Analysis and Existing Solutions

We first briefly formulate our objectives (see Section 5.3.1) and afterwards discuss corresponding related work (see Section 5.3.2 and Table 5.1).

5.3.1 Objectives and Threat Model

In this paper, we aim to combine the security benefits of OS extensions with the deployability benefits of application layer solutions. We identify the following objectives:

- O1 No firmware modification:** The solution does not rely on or require customized Android firmware, such as extensions to Android’s middleware, kernel or the default configuration files (e.g., policy files), and is able to run on stock Android versions. This also excludes availability of root privileges, since root can only be acquired through a firmware modification on newer Android versions due to increasingly stringent SELinux policies.
- O2 No app modification:** The solution does not rely on or require any modifications of monitored apps’ code, such as rewriting existing code.
- O3 Robust reference monitor:** The solution provides a robust reference monitor. This encompasses: 1) the presence of a strong security boundary, such as a process boundary, between the reference monitor and untrusted code; and 2) the monitor cannot be bypassed, e.g., using a code representation that is not monitored, such as native code.
- O4 Secure isolation of untrusted code:** This objective encompasses fail-safe defaults and complete mediation by the reference monitors. The solution provides a reference monitor that mediates all interaction between the untrusted code and the Android system, or, in case no complete mediation can be established, enforces fail-safe defaults that isolate the app on non-mediated channels in order to prevent untrusted code from escalating its privileges.

Threat model. We assume that the Android OS is trusted, including the Linux kernel and the Android application framework. This includes the assumption that an application cannot compromise the integrity of the kernel or application framework at runtime. If the kernel or application framework were compromised, no security guarantees could be upheld. Protecting the kernel and framework integrity is an orthogonal research direction for which different approaches already exist, such as trusted computing, code hardening, or control flow integrity.

Furthermore, we assume that untrusted third-party applications have full control over their process and the associated memory address space. Hence the attacker can modify its app’s code at runtime, e.g., using native code or Java’s reflection interface.

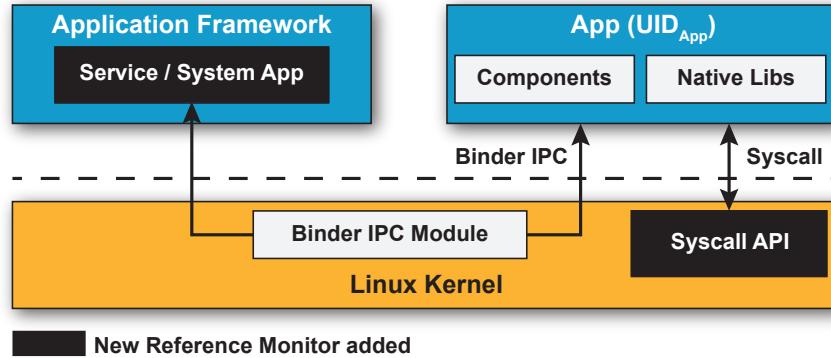


Figure 5.2: Instrumentation points for operating system security extensions.

| Objectives | OS ext. | IRM | Sep. app | Boxify |
|--|---------|-----|----------|--------|
| O1: No system modification | ✗ | ✓ | ✓ | ✓ |
| O2: No application modification | ✓ | ✗ | ✗ | ✓ |
| O3: Robust reference monitor | ✓ | ✗ | ✓ | ✓ |
| O4: Secure isolation of untrusted code | ✓ | ✗ | ✗ | ✓ |

✓ = applies; ✗ = does not apply.

Table 5.1: Comparison of deployment options for Android security extensions based on desired objectives.

5.3.2 Existing Solutions

We systematically analyze prior solutions on app sandboxing.

5.3.2.1 Android Security Extensions

Many improvements to Android’s security model have been proposed in the literature, addressing a variety of shortcomings in protecting the end-user’s privacy. In terms of deployment options, we can distinguish between solutions that extend the Android OS and solutions that operate at the application layer only.

Operating system extensions. The vast majority of proposals from the literature (e.g. [30, 62, 64, 12, 23, 86]) statically enhance Android’s application framework and Linux kernel with additional reference monitors and policy decision points (see Figure 5.2). The proposed security models include, for instance, context-aware policies [23], app developer policies [64], or Chinese wall policies [12]. More recent approaches [74, 61, 83] avoid static changes to the OS by dynamically instrumenting core system services (like Binder and Zygote) or the Android bootup scripts in order to interpose [68] un-

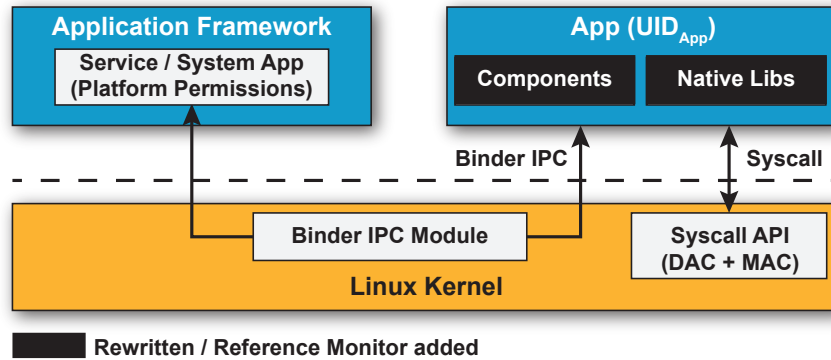


Figure 5.3: Instrumentation points for application code rewriting and inlining reference monitors.

trusted apps' syscalls and IPC. Since in all approaches the reference monitors are part of the application framework and kernel, there inherently exists a strong security boundary between the reference monitor and untrusted code (O3: ✓). Moreover, this entails that these reference monitors are by design part of the callee-side of all interaction of the untrusted app's process with the system and cannot be bypassed (O4: ✓). On the downside, these solutions require modification of the Android OS (image) or root privileges to be deployed (O1: ✗; O2: ✓).

Additionally, a number of solutions exist that particularly target higher-security deployments [10, 73, 55, 3], such as government and enterprise. Commercial products exist that implement these solutions in the form of tailored mobile platforms (e.g., Blackphone¹, GreenHills², or Cryptophone³). These products target specialized user groups with high security requirements—not the average consumer—and are thus deployed on a rather small scale.

Application layer solutions. At the application layer, the situation for third-party security extensions is bleak. Android's UID-based sandboxing mechanism strictly isolates different apps installed on the same device. Android applications run with normal user privileges and cannot elevate to root in order to observe the behavior of other apps, e.g., like classical trace or anti-virus programs on desktop operating systems [42]. Also, Android does not offer any APIs that would allow one app to monitor or restrict the actions of another app at runtime. Only static information about other apps on the device is available via the Android API, i.e., application metadata, such as the package name or signing certificate, and the compiled application code and resources. Consequently, most commercially available security solutions are limited to *detecting* potentially malicious apps, e.g. by comparing metadata with predefined blacklists or by checking the application code for known malware signatures, but they

¹<https://blackphone.ch>

²<http://www.ghs.com/mobile/>

³<http://esdcryptophone.com>

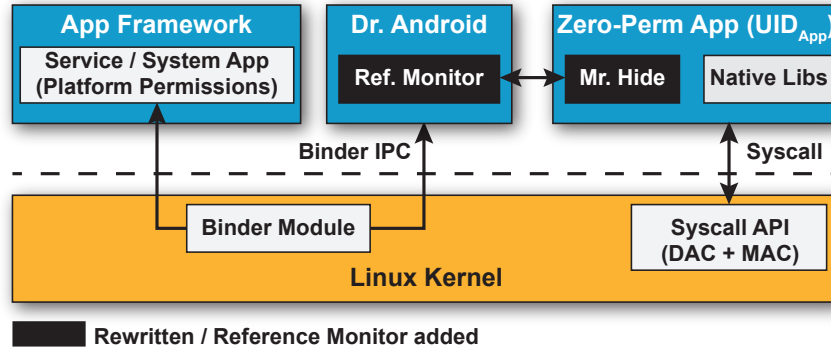


Figure 5.4: *Dr. Android and Mr. Hide* approach (54).

lack the ability to observe or influence the runtime behavior of other applications. As a result, their effectiveness is, at best, debatable [71, 97].

Few proposals in the academic literature [54, 88, 27, 70, P1] focus on application layer only solutions (see Figure 5.3). Existing systems mostly focus on access control by interposing security-sensitive APIs to redirect the control flow to an additionally inlined reference monitor within the app (e.g., Aurasium [88], I-ARM-Droid [27], RetroSkeleton [26], AppGuard [P1]). DroidForce [70] additionally pre-processes target apps with static data flow analysis to identify strategic policy enforcement points and to redirect policy decision making to a separate app.

All these systems are based on rewriting the application code to inline reference monitors or redirect control flows, which works without modifications to the firmware and is thus suitable for large-scale deployment (O1: ✓; O2: ✗). However, app rewriting causes security problems and also a couple of practical deployment problems. First, inlining the reference monitor within the process of the untrusted app itself might be suitable for “benign-but-buggy” apps; however, apps that actively try to circumvent the monitor will succeed as there exists no strong security boundary between the app and the monitor. In essence, this boils down to an arms race between hooking security critical functions and finding new ways to compromise or bypass the monitor [50], where currently native code gives the attacker the advantage (O3: ✗; O4: ✗). Moreover, re-writing application code requires re-signing of the app, which breaks Android’s signature-based same origin policy and additionally raises legal concerns about illicit tampering with foreign code. Lastly, re-written apps have to be reinstalled. This is not technically possible for pre-installed system apps; other apps have to be uninstalled in order to install a fresh, rewritten version, thereby incurring data loss.

Separate app. Dr. Android and Mr. Hide [54] (see Figure 5.4) is a variant of inlined reference monitoring (O1: ✓; O2: ✗) that improves upon the security of the reference monitor by moving it out of the untrusted app and into a separate app. This establishes a strong security boundary between the untrusted app and the reference monitor as they run in separate processes with different UIDs (O3: ✓). Additionally, it revokes

all Android platform permissions from the untrusted app and applies code rewriting techniques to replace well-known security-sensitive Android API calls in the monitored app with calls to the separate reference monitor app that acts as a proxy to the application framework. The benefit of this design is that in contrast to inlined monitoring, the untrusted, zero-permission app cannot gain additional permissions by tampering with the inlined/rewritten code. However, this enforcement only addresses the platform permissions. The untrusted app process still has a number of Linux privileges (such as access to the Binder interface or file system), and it has been shown that even a zero-permission app is still capable of escalating its privileges and violate the user's privacy [66, 44, 20, 13, 89, 60, 94, 95] (O4: ✗).

5.3.2.2 Sandboxing on traditional OSes

Restricting the access rights of untrusted applications has a longstanding tradition in desktop and server operating systems. Few solutions set up user-mode only sandboxes without relying on operating system functionality by making strong assumptions about the interface between the target code and the system (e.g., absence of programming language facilities to make syscalls or direct memory manipulation). Among the most notable user-space solutions are *native client* [92] to sandbox native code within browser extensions and the *Java virtual machine* [53] to sandbox untrusted Java applications.

Other solutions, which loosen the assumptions about the target interface to the system rely on operating system security features to establish process sandboxes. For instance, *Janus* [42], one of the earlier approaches, introduced an OS-supported sandbox for untrusted applications on Solaris 2.4, which was based on syscall monitoring and interception to restrict the untrusted process' access to the underlying operating system. The monitor was implemented as a separate process with necessary privileges to monitor and restrict other processes via the `/proc` kernel interface. Modern browsers like *Chromium* [82, 21, 81] employ different sandboxing OS facilities (e.g, `seccomp` mode) to mitigate the threat of web-based attacks against clients by restricting the access of untrusted code.

App virtualization. Sandboxing also plays a role in more recent *application virtualization* solutions [46, 85, 22, 59], where applications are transparently encapsulated into execution environments that replace (parts of) the environment with emulation layers that abstract the underlying OS and interpose all interaction between the app and the OS. App virtualization is currently primarily used to enable self-contained, OS-agnostic software, but also provides security benefits by restricting the interface and view the encapsulated app has of the system.

Similarly to these traditional sandboxes and in particular to app virtualization, BOXIFY forms a user-mode sandbox that builds on top of existing operating system facilities of Android. Thereby, it establishes app sandboxes that encapsulate Android apps without the need to modify the OS and without the need to make any assumptions

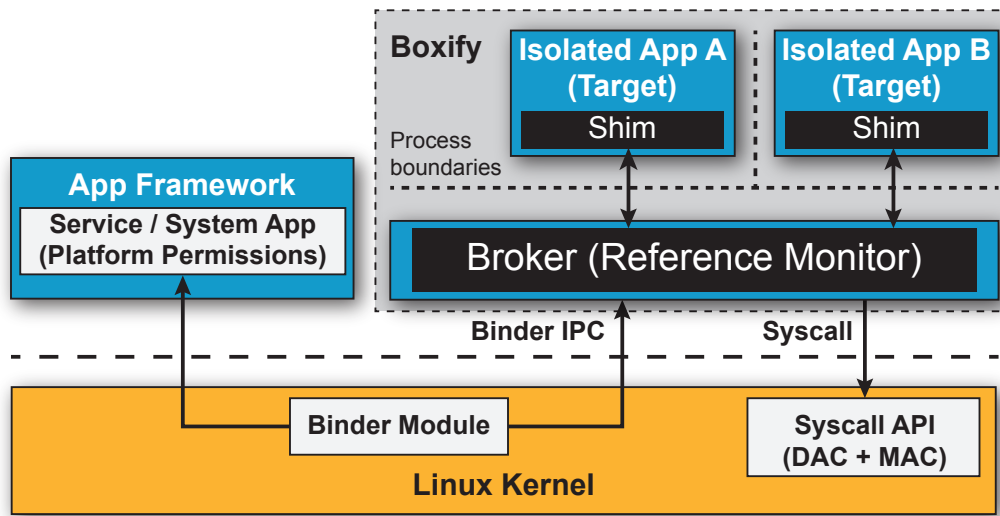


Figure 5.5: Architecture overview of BOXIFY.

about the apps' code.

5.4 Boxify Architecture

We present the BOXIFY design and implementation.

5.4.1 Design Overview

The key idea of BOXIFY is to securely sandbox Android apps, while avoiding any modification of the OS and untrusted apps. BOXIFY accomplishes this by dynamically loading and executing the untrusted app in one of its own processes. The untrusted application is not executed by the Android system itself, but runs completely encapsulated within the runtime environment that BOXIFY provides and that can be installed as a regular app on stock Android (see Figure 5.5). This approach eliminates the need to modify the code of the untrusted application and works without altering the underlying OS (O1: ✓; O2: ✓). It thus constitutes the first solution that ports the concept of app virtualization to the stock Android OS.

The primary challenge for traditional application sandboxing solutions is to completely mediate and monitor all I/O between the sandboxed app and the system in order to restrict the untrusted code's privileges. The key insight for our BOXIFY approach is to leverage the security provided by *isolated processes* in order to isolate the untrusted code running within the context of BOXIFY by executing it in a completely de-privileged process that has no platform permissions, no access to the Android middleware, nor the ability to make persistent changes to the file system.

However, Android apps are tightly integrated within the application framework, e.g.,

for lifecycle management and inter-component communication. With the restrictions of an isolated process in place, encapsulated apps are rendered dysfunctional. Thus, the key challenge for BOXIFY essentially shifts from constraining the capabilities of the untrusted app to now gradually permitting I/O operations in a controlled manner in order to securely re-integrate the isolated app into the software stack. To this end, BOXIFY creates two primary entities that run at different levels of privilege: A privileged controller process known as the **Broker** and one or more isolated processes called the **Target** (see Figure 5.5).

The **Broker** is the main BOXIFY application process and acts as a mandatory proxy for all I/O operations of the **Target** that require privileges beyond the ones of the isolated process. Thus, if the encapsulated app bypasses the **Broker**, it is limited to the extremely confined privilege set of its isolated process environment (*fail-safe defaults*; O4: ✓). As a consequence, the **Broker** is an ideal control-flow location in our BOXIFY design to implement a reference monitor for any privileged interaction between a **Target** and the system. Any syscalls and Android API calls from the **Target** that are forwarded to the **Broker** are evaluated against a security policy. Only policy-enabled calls are then executed by the **Broker** and their results returned to the **Target** process. To protect the **Broker** (and hence reference monitor) from malicious app code, it runs in a separate process under a different UID than the isolated processes. This establishes a strong security boundary between the reference monitor and the untrusted code (O3: ✓). To transparently forward the syscalls and Android API calls from the **Target** across the process boundary to the **Broker**, BOXIFY uses Android's Binder IPC mechanism. Finally, the **Broker**'s responsibilities also include managing the application lifecycle of the **Target** and relaying ICC between a **Target** and other (**Target**) components.

The **Target** hosts all untrusted code that will run inside the sandbox. It consists of a shim that is able to dynamically load other Android applications and execute them. For the encapsulated app to interact with the system, it sets up interceptors that interpose system and middleware API calls. The interceptors do *not* form a security boundary but establish a compatibility layer when the code inside the sandbox needs to perform otherwise restricted I/O by forwarding the calls to the **Broker**. All resources that the **Target** process uses have to be acquired by the **Broker** and their handles duplicated into the **Target** process.

By encapsulating untrusted apps and interposing all their (privileged) I/O operations, BOXIFY is able to effectively enforce security- and privacy-protecting policies. Based on syscall interposition, BOXIFY has fine-grained control over network and filesystem operations. Intercepting Binder IPC enables the enforcement of security policies that were so far only achievable for OS extensions, but at application layer only.

Moreover, with this architecture, BOXIFY can provide a number of interesting novel features. BOXIFY is capable of monitoring multiple (untrusted) apps at the same time. By creating a number of **Target** processes, multiple apps can run in parallel yet securely isolated in a single instance of BOXIFY. Since the **Broker** fully controls all

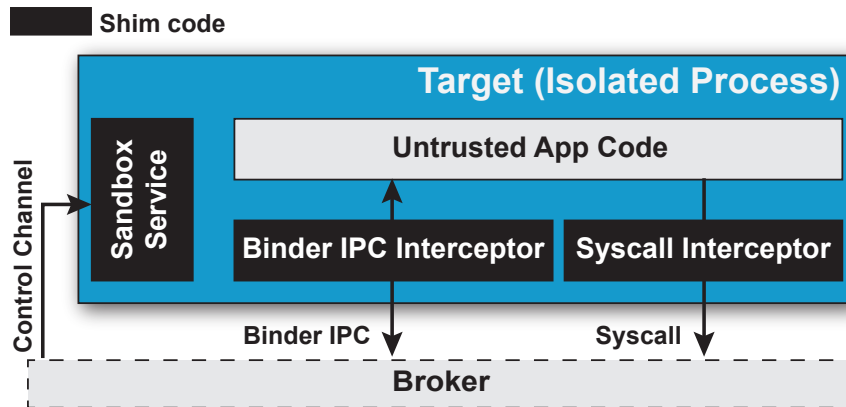


Figure 5.6: Components of a Target process.

inter-component communication between the sandboxed apps, it is able to not only separate different apps from one another but also to allow controlled collaboration between them. Further, BOXIFY has the ability to execute apps that are not regularly installed on the phone: Since BOXIFY executes other apps by dynamically loading their code into one of its own processes and handles all the interaction between the sandboxed application and the OS, there is no need to register the untrusted app with the Android system. Hence, applications can be installed into, updated, or removed from BOXIFY without involving the `PackageInstaller` or having system privileges. A potential application of these features are application containers (e.g., enterprise app domain, see Section 5.5.4).

5.4.2 Target

The **Target** process contains four main entities (see Figure 5.6): The **SandboxService** (1) provides the **Broker** with a basic interface for starting and terminating apps in the sandbox. It is also responsible for setting up the interceptors for Binder IPC (2) and syscalls (3), which transparently forward calls issued by the untrusted application to the **Broker**.

1) SandboxService. Isolated processes on Android are realized as specifically tagged Service components (see Section 5.2). In BOXIFY each **Target** is implemented as such a tagged **SandboxService** component of the BOXIFY app. When a new **Target** should be spawned, a new, dedicated **SandboxService** is spawned. The **SandboxService** provides an IPC interface that enables the **Broker** to communicate with the isolated process and to call two basic lifecycle operations for the **Target**: `prepare` and `terminate`. The **Broker** invokes the `prepare` function to initialize the sandbox environment for the execution of a hosted app. As part of this preparation, the **Broker** and **Target** exchange important configuration information for correct operation of the **Target**, such as app meta-information and Binder IPC handles that allow bi-directional IPC between

Broker and **Target**. The `terminate` function shuts down the application running in the sandbox and terminates the **Target** process.

The biggest technical challenge at this point was “*How to execute another third-party application within the running isolated service process?*” Naïvely, one could consider, for instance, a warm-restart of the app process with the new application code using the `exec` syscall. However, we discovered that the most elegant and reliable solution is to have the **Broker** initially imitate the `ActivityManager` by instructing the **Target** process to load (i.e., *bind*) another application to its process and afterwards to relay any lifecycle events between the actual application framework and the newly loaded application in the **Target** process. The *bind* operation is supported by the standard Android application framework and used during normal app startup. The exact procedure is illustrated in Figure 5.7. The **Broker** first creates a new **SandboxService** process (❶), which executes with the privileges of an isolated process. This step actually involves multiple messages between the **Broker** process, the **Target** process and the system server, which we omitted here for the sake of readability. As a result, the **Broker** process receives a Binder handle to communicate with the newly spawned **SandboxService**. Next, the **Broker** uses this handle to instruct the **SandboxService** to prepare the loading of a sandboxed app (❷) by setting up the Binder IPC interceptor and syscall interceptor (using the meta-information given as parameters of the `prepare` call). The **SandboxService** returns the Binder handle to its `ApplicationThread` to the **Broker**. The application thread is the main thread of a process containing an Android runtime and is used by the `ActivityManager` to issue commands to Android application processes. At this point, the **Broker** emulates the behavior of the `ActivityManager` (❸) by instructing the `ApplicationThread` of the **Target** with the `bindApplication` call to load the target app into its Android runtime and start its execution. By default, it would be the `ActivityManagerService` as part of the application framework that uses this call to instruct newly forked and specialized `Zygote` processes to load and execute an application that should be started. After this step, the sandboxed app is executing.

As an example how a sandboxed app can be used, we briefly explain how an `Activity` component of the sandboxed app can be launched, e.g., as result of clicking its entry in a launcher. As explained in Section 5.4.3, the **Virtualization Layer** creates a mapping from generic BOXIFY components to **Target** components. In this case, it maps the `Activity` component of **Target** to an `Activity` component of BOXIFY. The **Broker** requests the `Activity` launch from the `ActivityManager` in the `SystemServer` (❹), which allocates the required resources. After allocation, it schedules the launch of the `Activity` component by signaling the `ApplicationThread` of the targeted app (❺), which in this case is the BOXIFY app. Thus, the **Virtualization Layer** resolves the targeted `Activity` component and relays the signal to the corresponding **Target** process (❻).

2) Binder IPC Interceptor. Android applications use the Binder IPC mechanism to communicate with the (remote) components of other applications, including the application framework services and apps. In order to interact via Binder IPC with a remote

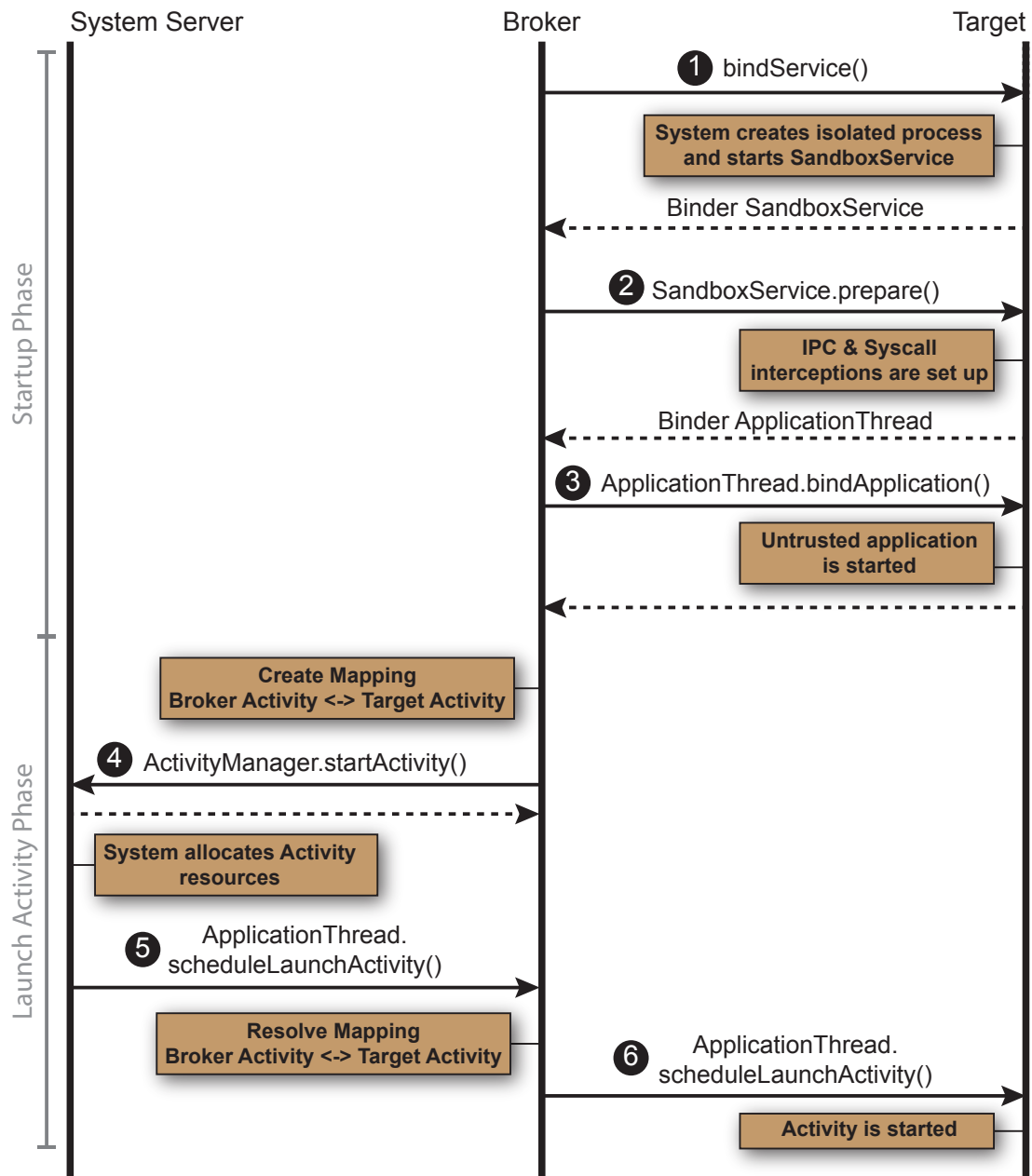


Figure 5.7: Process to load an app into a Target process and to launch one of its Activities.

component, apps must first acquire a Binder handle that connects them to the desired component. To retrieve a Binder handle, applications query the `ServiceManager`, a central service registry, that allows clients to lookup system services by their common names. The `ServiceManager` is the core mechanism to bootstrap the communication of an application with the Android application framework. Binder handles to non-system services, such as services provided by other apps, can be acquired from the core framework services, most prominently the `ActivityManager`.

BOXIFY leverages this choke point in the Binder IPC interaction to efficiently intercept calls to the framework in order to redirect them to the **Broker**. To this end, BOXIFY replaces references to the `ServiceManager` handle in the memory of the **Target** process with references to the Binder handle of the **Broker** (as provided in the `prepare` function). These references are constrained to a few places and can be reliably modified using the Java Reflection API and native code. Consequently, all calls directed to the `ServiceManager` are redirected to the **Broker** process instead, which can then manipulate the returned Binder objects in such a way that any subsequent interactions with requested services are also redirected to the **Broker**. Furthermore, references to a few core system services, such as the `ActivityManager` and `PackageManager`, that are passed by default to new Android app runtimes, need to be replaced as well. By modifying only a small number of Binder handles, BOXIFY intercepts all Binder IPC communication. The technique is completely agnostic of the concrete interface of the redirected service and can be universally applied to all Binder interactions.

3) Syscall Interceptor. For system call interception, we rely on a technique called `libc` hooking (used, for instance, also in [88]). On Android, libraries or *shared objects* like `libc` are relocatable ELF files that are mapped into a process' address space when loaded. Android applications and shared objects shipped with Android are dynamically linked against the Bionic `libc` library to avoid code duplication and conserve memory. In contrast to static linking, where addresses of library functions are fixed at compile time, dynamically linked shared objects can be loaded at an arbitrary location in memory. Hence, the addresses of symbols defined in such a shared object need to be dynamically resolved at runtime. To this end, ELF files that are dynamically linked to some shared object do not call the functions defined in the shared object directly, but instead jump to a stub function in the ELF's procedure linkage table (PLT). This stub function retrieves the real target address of the library function from the ELF's global offset table (GOT) and then branches to it. In other words, the ELF's global offset table contains an array of function pointers of all dynamically linked external functions referenced by its code. `Libc` hooking leverages this level of indirection introduced by dynamic linking to efficiently intercept calls to `libc` functions: it suffices to iterate over every loaded ELF file and replace the functions pointers in its global offset table by pointers to an alternative implementation. Using this technique, we efficiently intercept calls to `libc` functions and redirect these calls to a service client running in the **Target** process. This client forwards the function calls via IPC to a custom service component running in the **Broker**.

In contrast to the IPC interception, which redirects all IPC communication to the **Broker**, the syscall interception is much more selective about which calls are forwarded: We do not redirect syscalls that would be anyway granted to an isolated process, because there is no security benefit from hooking these functions: a malicious app could simply remove the hook and would still succeed with the call. This exception applies to calls to read world-readable files and to most system calls that operate purely on file descriptors

(e.g. read, write). Naturally, by omitting the indirection via our **Broker**, these exempted calls perform with native performance. However, **BOXIFY** still hooks calls that are security-critical and that are not permitted for isolated processes, such as system calls to perform file system operations (e.g. open, mkdir, unlink) and network I/O (socket, getaddrinfo). For a few calls, such as file operations, whose success depends on the given parameter, the syscall interception is parameter-sensitive in its decision whether or not to forward this operation to the **Broker**.

5.4.3 Broker

The **Broker** is the main application process of **BOXIFY** and is thus not subject to the restrictions imposed by the isolated process. It holds all platform permissions assigned to the **BOXIFY** app and can normally interact with the Android middleware. The **Broker** acts as a mandatory proxy for all interactions between the **Target** processes and the Android system and thus embodies the reference monitor of **BOXIFY**. These interactions are bi-directional: On the one hand, the untrusted app running in the **Target** process issues IPC and syscalls to the system; on the other hand, the Android middleware initiates IPC calls to **Target** (e.g., basic lifecycle operations) and the **Broker** has to dispatch these events to the correct **Target**.

The **Broker** is organized into three main layers (see Figure 5.8): The **API Layer** (4) abstracts from the concrete characteristics of the Android-internal IPC interfaces to provide compatibility across different Android versions. It bridges the semantic gap between the raw IPC transactions forwarded by the **Target** and the application framework semantics of the **Core Logic Layer** (5), which implements the fundamental mechanics of the virtual runtime environment that **BOXIFY** provides. All interaction with the system happens through the **Virtualization Layer** (6), which translates between the virtual environment inside of **BOXIFY** and the Android system on the outside. In the following, we will look at every layer in more detail.

4) API Layer. The **API Layer** is responsible for receiving and unwrapping the redirected syscall parameters from the Syscall Interceptor in the **Target** and relaying them to the **Core Logic Layer** for monitoring and execution. More importantly, it transforms the raw Binder IPC parcels received from the IPC Interceptor into a representation agnostic of the Android version.

In order to (efficiently) sandbox applications at the Binder IPC boundary, **BOXIFY** must semantically interpret the intercepted Binder parcels. However, intercepted parcels are in a raw representation that consists only of native types that the kernel module supports and the sender marshalled all higher-level objects (e.g., Java classes) to this representation. This impedes an efficient sandboxing. To solve this problem, **BOXIFY** leverages the default Android toolchain for implementing Binder-based RPC protocols: To ensure that sender and receiver can actually communicate with each other, the receiver must know how to unmarshal the raw parcel data (exactly like **BOXIFY**).

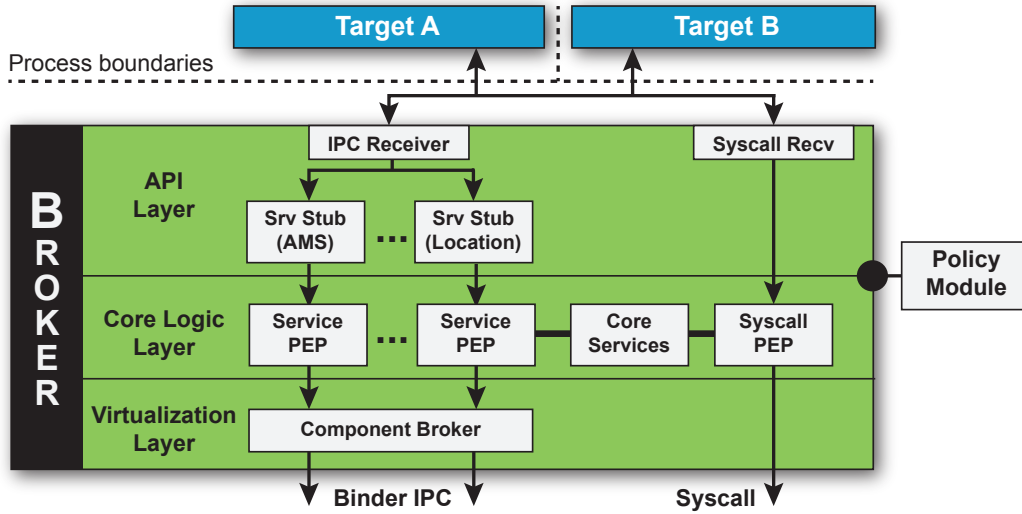


Figure 5.8: Architecture of the Broker.

Android supports the developers in this process through the *Android Interface Definition Language* (AIDL), which allows definitions of Binder interfaces very similar to Java interfaces, including the names and signatures of remotely callable functions. The Android SDK toolchain generates the required boilerplate marshalling code from AIDL definitions both for the receiver (*Stub*) and the sender (*Proxy*). For system services, these Stubs are automatically generated during system build and BOXIFY uses the generated Stubs (which ship with Android OS and are conveniently accessible to third-party application) to unmarshal the raw Binder IPC parcel back to their application framework semantic (i.e., Java objects, etc). In essence, this allows us to generate the API layer of the **Broker** in an almost fully-automatic way for each Android version on which BOXIFY is deployed. Since BOXIFY is in full control of the Binder handles of the encapsulated app (i.e., calls to the `ServiceManager`, `ActivityManager`, etc.), it can efficiently determine which Binder handle of the app addresses which system service and hence which Stub must be used to correctly unmarshal the raw Binder parcel intercepted from each handle.

However, the exact structure of the unmarshalled data and the functions (name and signature) depend entirely on the AIDL file. Since the system service interfaces describe the internal Android API, these interfaces change frequently between Android versions. Hence BOXIFY would have to implement each possible version of a Stub for every available Android version. Since this Stub implementation, in contrast to the marshalling logic, can not be automated, this complicates efficient sandboxing of apps across multiple Android versions. Consequently, it is desirable to transform the unmarshalled IPC data into a version-agnostic representation and then implement each Stub once and for all for this version. To accomplish this in BOXIFY, we borrow ideas from Google's proprietary *SafeParcel* class: In contrast to the regular Binder parcel,

```
/core/java/android/view/accessibility/IAccessibilityManager.aidl
/core/java/android/accounts/IAccountManager.aidl
/core/java/android/app/IActivityManager.java
/media/java/android/media/IAudioService.aidl
/core/java/android/app/backup/IBackupManager.aidl
/core/java/android/content/IClipboard.aidl
/core/java/android/net/IConnectivityManager.aidl
/core/java/android/content/IServiceManager.aidl
/core/java/android/hardware/input/IInputManager.aidl
/core/java/com/android/internal/view/IInputMethodManager.aidl
/location/java/android/location/ILocationManager.aidl
/core/java/android/os/storage/IMountService.java
/core/java/android/nfc/INfcAdapter.aidl
/core/java/android/app/INotificationManager.aidl
/core/java/android/content/pm/IPackageManager.aidl
/core/java/android/os/IPowerManager.aidl
/core/java/android/os/IServiceManager.java
/voip/java/android/net/sip/ISipService.aidl
/telephony/java/com/android/internal/telephony/ITelephonyRegistry.aidl
/core/java/android/hardware/usb/IUsbManager.aidl
/core/java/android/app/IWallpaperManager.aidl
/core/java/android/os/IVibratorService.aidl
/wifi/java/android/net/wifi/IWifiManager.aidl
/core/java/android/view/IWindowManager.aidl
/core/java/android/net/nsd/INsdManager.aidl
```

Figure 5.9: Android internal APIs considered in our evaluation

the `SafeParcel` carries structural information about the data stored in it, which allows the receiver of an IPC request to selectively read parts of the payload without knowing its exact structure. We achieve the same result by transforming the version-dependent parcel into a version-agnostic key-value store (where keys are the parameter names of methods declared in the interface definitions) and adapting the **Core Logic Layer** and **Stub** implementations to work with these version-agnostic data stores. Thus, while the **API layer** is version-dependent and automatically generated for each Android version, the remaining layers of **Broker** are version-agnostic and implemented only once.

To illustrate the need for a version abstraction such as the **API Layer**, we evaluated the changes of the internal API between minor versions of the Android OS. We used the interface descriptions of the Android core system services as basis for our evaluation (see Listing 5.9).

Figure 5.10 summarizes our results and illustrates the differences of the API between two consecutive minor versions. We measured the *methods added (MA)*, *methods removed (MR)*, *parameters added (PA)*, *parameters removed (PR)*, *parameters type changed (PTC)*, and *method return type changed (MRTC)*. Not surprisingly the biggest changes occurred on a jump to a new major version. Although the cumulative changes are moderate, our statistics illustrate the unreliability of the *internal* API and any solution relying on the internal API has to cope with this unreliability. Thus, it is preferable if any version dependent code can be automatically generated.

5) Core Logic Layer. The **Core Logic Layer** provides essential functionality re-

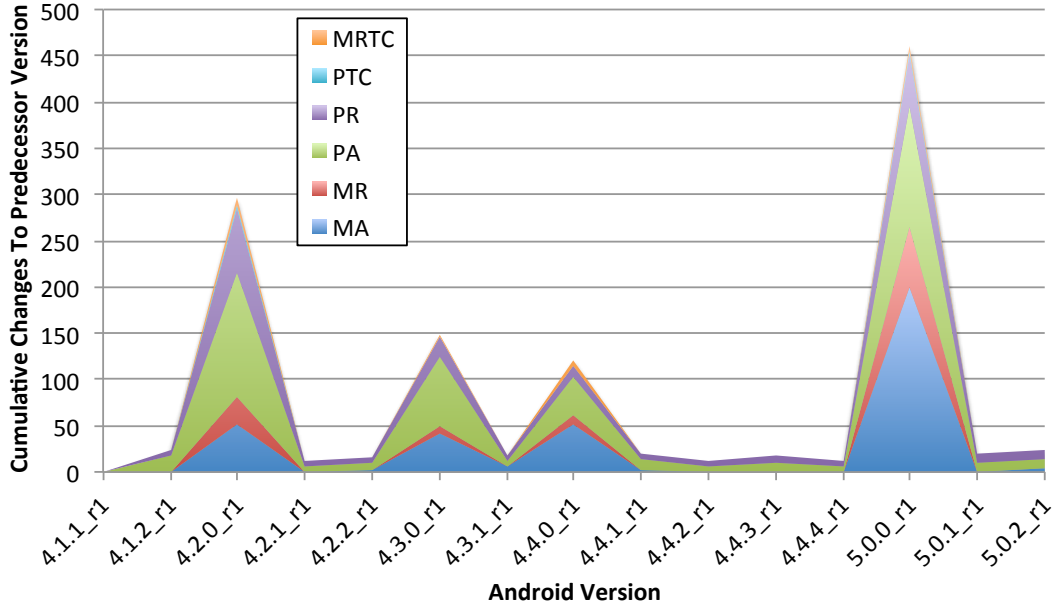


Figure 5.10: Internal API stability between minor Android versions.

quired to run apps on Android by replicating a small subset of the functionality that Android’s core system services provide. Most prominently, this layer provides a minimal implementation of the `PackageManager`, which manages the packages installed into the BOXIFY environment. Every call to a system service that is not emulated by the Core Logic Layer is passed on to the Virtualization Layer and thus to the underlying Android system. Other system services, such as the `LocationManager`, which are not necessarily required, can be instantiated at this layer as well, in case encapsulated apps are supposed to use the local, BOXIFY service implementation instead of the pristine Android service (e.g., servicing sandboxed apps with fake location data [98]). Hence, this layer decides whether an Android API call is emulated using a replicated service or forwarded to the system (through the Virtualization Layer). This layer is therefore responsible for managing the IPC communication between different sandboxed apps (abstractly like an “*ICC switch*”).

Furthermore, the Core Logic Layer implements the policy enforcement points (PEP) for Binder IPC services and syscalls. Because the API Layer already bridges the semantic gap between kernel-level IPC and Android application framework semantics, this removes the burden for dealing with low-level semantics in the IPC PEPs. We emulate the integration of enforcement points into pristine Android services by integrating these points into our mandatory service proxies. This allows us to instantiate security models from the area of OS security extensions (see Section 5.3.2), but at the application layer. One default security model that BOXIFY provides is the permission enforcement and same origin model of Android. For instance, the replicated `ActivityManager` will enforce permissions on calls between components of two sandboxed apps. We present

further security models from related work on OS security extensions that we integrated at this layer in Section 5.5.4 and for future work we consider a programmable interface for extending **Core Logic Layer** security in the spirit of ASM [51] and ASF [S1]. For calls that are not protected by a permission, the **Broker** can also choose to enable direct communication between the target app and the requested Android system service. This can improve performance for non-critical services such as the `SurfaceFlinger` (for GUI updates) at the cost of losing the ability to mediate calls to these services.

The syscall PEP enforces system call policies in the spirit of [68] with respect to network and filesystem operations. Its responsibilities are twofold: First, it functions as a transparent compatibility layer by emulating the file-system structure of the Android data partition (e.g., `chroot` of sandboxed apps by emulating a home directory for each sandboxed app⁴ within the home directory of the BOXIFY app). Second, it emulates the access control of the Linux kernel, i.e., compartmentalization of sandboxed apps by ensuring that they cannot access private files of other apps as well as enforcing permissions (e.g., preventing a sandboxed app without Internet permission from creating a network socket).

6) Virtualization Layer. The sandbox environment must support communication between sandboxed apps and the Android application framework, because certain system resources cannot be efficiently emulated (e.g., `SurfaceFlinger` for GUI) or not emulated at all (e.g., hardware resources like the camera). However, the sandbox must be transparent to the **Target** and all interaction with the application framework must appear as in a regular app. At the same time, the sandbox must be completely opaque to the application framework and sandboxed apps must be hidden from the framework; otherwise, this leads to inconsistencies that the framework considers as runtime (security) exceptions.

In BOXIFY, the **Virtualization Layer** is responsible for translating the bi-directional communication between the Android application framework and the **Target**. It achieves the required semi-transparent communication with a technique that can be abstractly described as “*ICC Network Address Translator*”: On outgoing calls from **Target** to framework, it ensures that all ICC appears as coming from the BOXIFY app instead of the sandboxed app. As described earlier, all Binder handles of a **Target** are substituted with handles of the **Broker**, which relays the calls to the system. During relay of calls, the **Virtualization Layer** manipulates the call arguments to hide components of sandboxed apps by substituting the component identifiers with identifiers of components of the BOXIFY app. On incoming calls from the framework, the **Virtualization Layer** substitutes the addressed BOXIFY component with the actually addressed component of the sandboxed app and dispatches the call. In order to correctly substitute addressed components, the **Virtualization Layer** maintains a mapping between **Target** and BOXIFY component names, or in case the **Target** component is not addressed by a name but

⁴Recall that sandboxed apps are not installed in the system but only in the BOXIFY environment, and hence do not have a native home directory.

a Binder handle that was given prior to the framework, the mapping is between the released Binder handle and its owning **Target** component.

A concrete example where this technique is applied is requesting the launch of a **Target** Activity component from the application framework (see Figure 5.7). The **Virtualization Layer** substitutes the Activity component with a generic Activity component of BOXIFY if a call to the `ActivityManager` occurs. When the service calls back for scheduling the Activity launch, the **Virtualization Layer** dispatches the scheduling call to the corresponding **Target** Activity component.

Lastly, we hook the application runtime of BOXIFY's **Broker** process (using a technique similar to [P4]) in order to gain control over the processing of incoming Binder parcels. This enables the **Broker** to distinguish between parcels addressed to BOXIFY itself and those that need to be forwarded to the **Target** processes.

5.4.4 System Integration

Lastly, we discuss some aspects of integrating sandboxed apps into the default application framework.

Launcher. Since sandboxed apps have to be started through BOXIFY (and are not regularly installed on the system), they cannot be directly launched from the default launcher. A straightforward solution is to provide a custom launcher with BOXIFY in form of a dedicated Activity. Alternatively, BOXIFY could register as a launcher app and then run the default launcher (or any launcher app of the user's choice) in the sandbox, presenting the union of the regularly installed apps and apps installed in the sandbox environment; or BOXIFY launcher widgets could be placed on the regular home screen to launch sandboxed apps from there.

App stores. Particularly smooth is the integration of BOXIFY with app store applications, such as the Google Play Store. Since no special permissions are required to install apps into the sandbox, we can simply run the store apps provided by Google, vendors, and third-parties in BOXIFY to install new apps there. For example, clicking install in the sandboxed Play Store App will directly install the new app into BOXIFY. Furthermore, Play Store (and vendor stores) even take care of automatically updating all apps installed in BOXIFY, a feature that IRM systems have to manually re-implement.

Statically registered resources. Some resources of apps are statically registered in the system during app installation. Since sandboxed apps are not regularly installed, the system is unaware of their resources. This concerns in particular Activity components that can receive Intents for, e.g., content sharing, or package resources like icons. However, some resources like Broadcast Receiver components can be dynamically registered at runtime and BOXIFY uses this as a workaround to dynamically register the Receivers declared statically in the Manifests of sandboxed apps.

Table 5.2: Micro-benchmarks middleware (200 runs)

| API Call | Native | on BOXIFY | Overhead |
|-----------------|-----------|-----------|-----------------|
| Open Camera | 103.24 ms | 104.48 ms | 1.24ms (1.2%) |
| Query Contacts | 7.63 ms | 8.55 ms | 0.92 ms (12.0%) |
| Insert Contacts | 66.49 ms | 67.51 ms | 1.02 ms (1.5%) |
| Delete Contacts | 75.86 ms | 76.81 ms | 0.95 ms (0.9%) |
| Create Socket | 120.83 ms | 121.58 ms | 0.75 ms (0.6%) |

Table 5.3: Micro-benchmarks syscalls (15k runs)

| Libc Func. | Native | on BOXIFY | Overhead |
|------------|--------------|---------------|---------------|
| create | 47.2 μ s | 162.4 μ s | 115.2 μ s |
| open | 9.5 μ s | 122.7 μ s | 113.2 μ s |
| remove | 49.5 μ s | 159.6 μ s | 110.1 μ s |
| mkdir | 88.4 μ s | 199.4 μ s | 111.0 μ s |
| rmdir | 71.2 μ s | 180.7 μ s | 109.5 μ s |

5.5 Evaluation

We discuss the prototypical implementation of BOXIFY in terms of performance impact, security guarantees, and app robustness, and present concrete use-cases of BOXIFY. Our prototype comprises 11,901 lines of Java code, of which 4,242 LoC are automatically generated (API Layer), and 3,550 lines of additional C/C++ code. All tests described in the following were performed on an LG Nexus 5 running Android 4.4.4, which is currently the most widely used version in the Android ecosystem.

5.5.1 Performance Impact

To evaluate the performance impact of BOXIFY on monitored apps, we compare the results of common benchmark apps and of custom micro-benchmarks for encapsulated and native execution of apps.

Table 5.2 and Table 5.3 present the results of our micro-benchmarks for common Android API calls and for syscall performance. Intercepting calls to the application framework imposes an overhead around 1%, with the exception of the very fast Query Contacts (12%). For syscalls, we measured the performance of calls that request file descriptors for file I/O in private app directories (or external storage) and that are proxied by the **Broker**. We observe a constant performance overhead of $\approx 100\mu$ s, which corresponds to the required time of the additional IPC round trip for the communication with the **Broker** on our test platform. However, the syscall benchmarks depict a worst-case estimation: The overall performance impact on apps is much lower, since high-frequent follow up operations on acquired file descriptors (e.g., read/write) need not to be intercepted and therefore run with native speed. We measured the overall performance penalty by executing several benchmarking apps on top of BOXIFY, which

Table 5.4: Benchmark tools (10 runs)

| Tool | Native | on BOXIFY | Loss |
|------------------|-----------|-----------|------|
| CF Bench v1.3 | 16082 Pts | 15376 Pts | 4.3% |
| Geekbench v3.3.1 | 1649 Pts | 1621 Pts | 1.6% |
| PassMark v1.0.4 | 3674 Pts | 3497 Pts | 4.8% |
| Quadrant v2.1.1 | 7820 Pts | 7532 Pts | 3.6% |

Table 5.5: Android versions supported by BOXIFY.

| Version | < 4.1 | 4.1 | 4.2 | 4.3 | 4.4 | 5.0 | 5.1 |
|-----------|----------------|-----|-----|-----|-----|-----|-----|
| Supported | ✗ [†] | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |

✓: supported; ✗: not supported

[†]: no *isolated process*

show an acceptable performance degradation of 1.6%–4.8% (see Table 5.4).

5.5.2 Runtime Robustness

To assess the robustness of encapsulated apps, we executed 1079 of the most popular, free apps from Google Play (retrieved in August 2014) on top of BOXIFY. For each sandboxed app we used the *monkeyrunner* tool⁵ to exercise the app’s functionality by injecting 500 random UI events. From the 1079 apps, 93 (8.6%) experienced a crash during testing. Manual investigation of the dysfunctional apps revealed that most errors were caused by apps executing exotic syscalls or rarely used Android APIs which are not covered by BOXIFY yet and thus fail due to the lack of privileges of the **Target** process (fail-safe defaults). This leads to a slightly lower robustness than reported for related work (e.g., [88, P1]) where bypassed hooks do not cause the untrusted app to crash but instead silently circumvent the reference monitor. The remaining issues were due to unusual application logic that relies on certain OS features (e.g., the process information pseudo-filesystem *proc*), which the current prototype of BOXIFY does not yet support. However, all of these are technical and not conceptual shortcomings of the current implementation of BOXIFY.

5.5.3 Portability

Table 5.5 summarizes the Android versions currently supported by our prototypical BOXIFY implementation. Our prototype supports all Android versions 4.1 through 5.1 and can be deployed on nine out of ten devices in the Android ecosystem [1]. Android versions prior to 4.1 are not supported due to the lack of the *isolated process* feature.

⁵http://developer.android.com/tools/help/monkeyrunner_concepts.html

5.5.4 Use-cases

BOXIFY allows the instantiation of different security models from the literature on Android security extensions. In the following, we present two selected use-cases on fine-grained permission control and domain isolation that have received attention before in the security community.

Fine-Grained Permission Control. The TISSA [98] OS extension empowers users to flexibly control in a fine-grained manner which personal information will be accessible to applications. We reimplemented the TISSA functionality as an extension to the Core Logic Layer of the BOXIFY Broker. To this end, we instrumented the mandatory proxies for core system services (e.g. `LocationManager`, `TelephonyService`) so that they can return a filtered or mock data set based on the user’s privacy settings. Users can dynamically adjust their privacy preferences through a management Activity added to BOXIFY. In total, the TISSA functionality required additional 351 lines of Java code to Core Logic Layer.

Domain Isolation. Particularly for enterprise deployments, container solutions have been brought forward to separate business apps from other (untrusted) apps [83, 10, 75].

We implemented a domain isolation solution based on BOXIFY by installing business apps into the sandbox environment. The Broker provides its own version of the `PackageManager` to directly deliver inter-component communication to sandboxed applications without involving the regular `PackageManager`, enabling controlled collaboration between enterprise apps while at the same time isolating and hiding them from non-enterprise apps and the OS.

To separate the enterprise data from the user’s private data, we exploit that the Broker is able to run separate instances of system services (e.g., `Contacts`, `Calendar`) within the sandbox. Our custom `ActivityManager` proxy now selectively and transparently redirects `ContentProvider` accesses by enterprise apps to the sandboxed counterparts of those providers.

Alternatively, the domain isolation concept described above was used to implement a privacy mode for end users, where untrusted apps are installed into a BOXIFY environment with empty (or faked) system `ContentProviders`. Thus, users can test untrusted apps in a safe environment without risking harm to their mobile device or private data. The domain isolation extension required 986 additional lines of code in the Core Logic Layer of BOXIFY.

5.5.5 Security Discussion

Our solution builds on *isolated processes* as fundamental security primitive. An isolated process is the most restrictive execution environment that stock Android currently has to offer, and it provides BOXIFY with better security guarantees than closest related

work [54]. In what follows, we identify different security shortcomings and discuss potential future security primitives of stock Android that would benefit BOXIFY and defensively programmed apps in general.

Privilege escalation. A malicious app could bypass the syscall and IPC interceptors, for instance, by statically linking `libc`. For IPC, this does not lead to a privilege escalation, since the application framework apps and services will refuse to cooperate with an isolated process. However, the kernel is unaware of the concept of an “isolated process” and will enforce access control on syscalls according to the process’ UID. Although the transient UIDs of isolated processes are very restricted in their filesystem access (i.e., only world readable/writable files), a malicious process has the entire kernel API as an attack vector and might escalate its privileges through a root or kernel exploit. In this sense, BOXIFY is not more secure than existing approaches that rely on the assumption that the stock Android kernel is hardened against root and kernel exploits.

To remedy this situation, additional layers of security could be provided by the underlying kernel to further restrict untrusted processes. This is common practice on other operating systems, e.g., on modern Linux distributions, where Chromium—the primary user of isolated process on Android—uses the *seccomp-bpf* facility to selectively disable syscalls of renderer processes and we expect this facility to become available on future Android versions with newer kernels. Similarly, common program tracing facilities could be made available in order to interpose syscalls more securely and efficiently [42, 68, 74].

Violating Least-Privilege Principle. The **Broker** must hold the union set of all permissions required by the apps hosted by BOXIFY in order to successfully proxy calls to the Android API. Since it is hard to predict a reasonable set of permissions beforehand, this means that the **Broker** usually holds all available permissions. This contradicts the principle of least privilege and makes the **Broker** an attractive target for the encapsulated app to increase its permission set. A very elegant solution to this problem would be a **Broker** that drops all unnecessary permissions. This resembles the privilege separation pattern [69, 84] of established Linux services like `ssh`, which drop privileges of sub-processes based on setting their UIDs, capabilities, or transitioning them to `seccomp` mode. Unfortunately, Android does not (yet) provide a way to *selectively* drop permissions *at runtime*.

Red Pill. Even though BOXIFY is designed to be invisible to the sandboxed app, it cannot exclude that the untrusted app gathers information about its execution environment that allow the app to deduce that it is sandboxed (e.g., checking its runtime UID or permissions). A malicious app can use this knowledge to change its runtime behavior when being sandboxed and thus hide its true intentions or refuse to run in a sandboxed environment. Prevention of this information leak is an arms race that a resolute attacker will typically win. However, while this might lead to refused functionality, it cannot be used to escalate the app’s privileges.

5.6 Conclusion

We presented the first application virtualization solution for the stock Android OS. By building on isolated processes to restrict privileges of untrusted apps and introducing a novel app virtualization environment, we combine the strong security guarantees of OS security extensions with the deployability of application layer solutions. We implemented our solution as a regular Android app called BOXIFY and demonstrated its capability to enforce established security policies without incurring significant runtime performance overhead.



Related Work

Since the release of Android in 2008, researchers have worked on improving and enhancing various of its security aspects. A major focus of their research has been overcoming the limitations of Android’s security model, in particular, its shortcomings in protecting the end-user’s privacy. The following chapter aims to give a broad overview of the approaches we consider relevant to our work. We classify these approaches by deployability, which has not only been one of the key motivations for our research in the first place but which we consider one of its distinguishing features. Hence, we differentiate between solutions extending the Android OS and solutions that operate solely at the application layer.

Operating system extensions Android’s inflexible and coarse-grained permission system inspired many researchers to propose extensions to the operating system. As early as 2009, Ongtang et al. [64] focused on the security requirements of smartphone applications and augmented the existing Android OS with a framework to meet them, by extending the permission system for inter-app communication. In their paper, they introduced a system called *Saint*, short for Secure Application INTeraction (Saint), a modified infrastructure that governs install-time permission assignment and their run-time use. Also focused on permissions was *Kirin*, a policy-based system proposed by Enck et al. [30] in 2009 that seeks to detect potential malware at installation time based on the permissions requested by an app. In 2010, Ongtang et al. introduced *Porscha* [63], a policy-based system for digital rights management on smartphones. In the same year, Nauman et al. [62] presented a modification of the Android software stack called *Apex* that enables dynamic permission revocations, while Conti et al. [23] went one step further with *CRePE*, a context-related policy enforcement mechanism to the Android software stack. In the following year, Grace et al. [44] were able to detect capability leaks on Android by analyzing phone images of different vendors, while Fraggaki et al. [39] presented *SORBET*, an external reference monitor approach to enforce coarse-grained secrecy and integrity policies. More recently, Wang et al. published *DeepDroid*, a system for enterprise policy enforcement via in-memory instrumentation of the system server, making use of our hooking technique for their instrumentation. With DALF [72], Raval et al. presented an extension to Android’s permission system that enables Android applications to act as policy plugins.

Other, more data-driven approaches were able to demonstrate the need for improvements to Android’s permission-based access control system. Barrera et al. [6] conducted an empirical analysis of Android’s permission system on 1,100 Android applications and, as a consequence, suggested refinements to its granularity. In a publication from 2011, Felt et al. [38] analyzed the effectiveness of application permissions making use of case studies on both Google Chrome extensions and Android apps. Using automated testing techniques, Felt et al. were further able to show that the mapping of permissions to APIs was only insufficiently documented [65]. Their analysis showed that roughly one-third of the tested applications were over-privileged, corroborating the hypothesis

that developers have difficulty implementing apps according to the principle of least privilege. Expanding on the work done by Felt et al., Backes et al. in 2016 published their analysis of Android’s permission specification [5], in which they revisited the permission to API mapping and analyzed policy enforcement point placement to advance the general understanding of Android’s application framework.

Some researchers focused on problems arising from inter-app communication on Android, most prominently the *confused deputy problem*, which was first highlighted by Davi et al. in 2010 [25]. This problem occurs when an app exposes permission protected functionality via an interface (intentionally or unintentionally) to an app without the permission, enabling privilege escalation attacks. To mitigate this problem, Felt et al. [66] modified the Android framework to inspect inter-process communication in order to reduce an app’s permissions if it is invoked from a less privileged application. In a similar vein, Dietz et al. presented *Quire* in 2011, [29] a system that allows app permissions to be dynamically assigned based on IPC call chains. Bugiel et al. [11] addressed privilege escalation attacks in a paper from 2011, in which they presented *XManDroid*, a security framework that detects covert channel attacks such as the Android trojan *Soundcomber* [77] They later extended their system to identify colluding apps [12], before they presented *Scippa* in 2014. *Scippa* is an extension to the Android IPC mechanism that provides provenance information to curb confused deputy attacks, even across multiple applications.

Another line of work leverages static and dynamic information flow techniques to detect privacy leaks in third-party apps. In 2009, Chaudhuri et al. introduced an operational semantics and type system for their Android app description language [16] that was later used as a basis for their static data flow analysis of Android applications [17]. The following year, Enck et al. presented the first approach for dynamic taint tracking on Android called *TaintDroid* [32], which enabled real-time monitoring of sensitive data flows. Expanding on the work done by Enck et al., Hornyack et al. [52] built a system called *AppFence* based on *TaintDroid*, which employs data shadowing and network connection blocking to protect against the exfiltration of sensitive data. With *AndroidLeaks*, Gibler et al. proposed a static analysis framework to detect privacy leaks in Android applications [40]. Zhou et al. developed *DroidRanger* [96], a tool for detecting Android malware in marketplaces based on behavioral footprinting and heuristics-based filtering.

While all these approaches were relevant to our work in their general objective of enhancing the security and privacy of the Android platform, they have in common that they cannot be deployed to stock Android devices, which forms a technological barrier for the average user. Modifying the firmware and platform code has some significant drawbacks: First, it requires rooting the device, which may void the user’s warranty. Furthermore, there exists not a single Android version but a plethora of vendor-specific variants that would need to be supported and maintained across OS updates.

Application Layer Solutions Several researchers have proposed security frameworks for third-party apps that do not rely on modifying the stock Android firmware. All of these frameworks make use of inlined reference monitors, which means that untrusted applications are rewritten to include a security monitor, the IRM.

Erlingsson and Schneider first formalized this concept in the development of the SASI/PoET/PSLang systems [36, 35] in which they implemented IRMs for x86 assembly code and Java bytecode in order to enforce custom security policies not supported by the Java virtual machine. Several other IRM implementations for Java followed. Notably, Polymer [8] is an IRM system based on edit automata that supports the composition of complex security policies from simple building blocks. In 2005, Chen et al. presented *Monitoring-Oriented Programming* (MOP) [19], a software development and analysis technique for Java that offers developers a rich set of tools to formally specify their policies. Hamlen et al.’s *SPoX* [48] establishes a formal connection between aspect-oriented programming and inlined reference monitoring wherein policy specifications denote aspect-oriented security automata. IRM systems have also been developed for other platforms. *Mobile* [49], for example, is an extension to Microsoft’s .NET Common Intermediate Language (CIL) that supports certified inlined reference monitoring, while the *S3MS.NET Run Time Monitor* [28] enforces security policies expressed in a variety of policy languages for .NET desktop and mobile applications on Windows phones.

Xu et al. first introduced the IRM technique to the Android platform in 2012 with *Aurasium* [88], a system that rewrites low-level function pointers of the libc library in order to intercept interactions between the application and the OS. Most of the functionality that is protected by Android’s permission system depends on such system calls and, thus, can be mediated at this level. However, the request’s semantics need to be recovered from the system calls’ low-level byte arrays in order to differentiate security-relevant from benign requests, which may be error-prone and break in the next version of Android at Google’s discretion. Jeon et al. [54] chose a different approach by placing the reference monitor into a separate application. Their system named *Dr. Android & Mr. Hide* allows removing all permissions from the monitored application as all calls to sensitive functionality are performed in the monitoring app. In contrast to Aurasium, it is fail-safe by default as it prevents both reflection and native code from executing such functionality. However, this enforcement only addresses platform permissions. The untrusted app process still has several Linux privileges (such as access to the Binder interface or file system), and it has been shown that even a zero-permission app is still capable of escalating its privileges and violate the user’s privacy [66, 44, 20, 13, 89, 60, 94, 95]. With *I-ARM-DROID* [27], Davis et al. present another IRM-based approach to enforce security policies, in which a set of security-sensitive API methods is identified by the user who can then accordingly specify and tailor their security policies to each application. While it supports the instrumentation of any Java method and covers reflective Java calls, their approach does not allow the instrumentation of applications directly on the device. Liu et al. [57] explored methods for efficient

privilege isolation for ad libraries in order to separate resource access permissions for ad libraries from that of the app logic, which they achieved via binary rewriting. You et al.’s Reference Hijacking [93], presented in 2016, can still be regarded as an IRM based approach. However, its use of the technique we proposed with BOXIFY for resetting app processes to replace system libs with instrumented versions indicates a blurring of the line between IRM and sandboxing. A different focus was chosen by Xue et al. when they presented *Malton* [91] in 2017. They were more concerned with on-device mobile malware analysis for which they used comprehensive in-process instrumentation of relevant APIs. *Malton* is a dynamic analysis tool that employs multi-layer monitoring, information flow tracking, and efficient path exploration to provide a holistic view of an app’s behavior. Most recently, Chandrashekhara et al. presented *Duvel* [15], a multi-profile manager for apps via storage sandboxing through bytecode instrumentation. *Duvel* builds on *BlueMountain* [14] by the same authors, which is a tool for app instrumentation that automatically integrates cloud storage services for Android apps.

The last field of research related to our work is concerned with application sandboxing and virtualization, which has a longstanding tradition in desktop and server operating systems. Solutions proposed by researchers from this field generally fall into two categories: They either set up user-mode only sandboxes without relying on operating system functionality. They achieve this by making strong assumptions about the interface between the target code and the system; for example, they assume the absence of programming language facilities to make syscalls or direct memory manipulation. Among the most notable user-space solutions are *native client* [92] to sandbox native code within browser extensions and the *Java virtual machine* [53] to sandbox untrusted Java applications. The alternative to making this assumption is to rely on operating system security features to establish process sandboxes. *Janus* [42], for example, one of the earlier approaches, introduced an OS-supported sandbox for untrusted applications on Solaris 2.4, which was based on system call monitoring and interception to restrict the untrusted process’ access to the underlying operating system. Modern browsers like *Chromium* [82, 21, 81] employ different sandboxing OS facilities to mitigate the threat of web-based attacks against clients by restricting the access of untrusted code. Sandboxing also plays a role in more recent *application virtualization* solutions [46, 85, 22, 59], where applications are executed in an isolated environment that (partially) emulates the underlying OS by interposing all interactions between the app and the OS.

We were the first to employ sandboxing and application virtualization on the Android platform with BOXIFY. Shortly after the publication of our work in 2015, Bianchi et al. presented *NJAS* [9], an alternative approach to application sandboxing on Android. Like BOXIFY, *NJAS* works by executing an application within the context of another one. However, they achieve sandboxing through system call interposition using *ptrace* while we rely on Android’s *isolated process* feature to deprive untrusted applications. *DroidPill* [90] from 2017 is a framework for malware creation built on BOXIFY’s idea of app virtualization. In their work, Chuan et al. present a technique called *app confusion*

attack that tricks users into interacting with a sandboxed clone of a benign app on their device. Finally, the authors of *SCLib*, a library to protect against component hijacking attacks, proposed deployment via BOXIFY in their 2018 paper [87].

7

Conclusion

This thesis started with the observation that Android’s set of security and privacy features – albeit its status as the most popular mobile operating system – used to be, and still mostly continues to be, quite limited. This is especially apparent when it comes to permissions for applications, which are usually dictated by the app developers, not chosen by the user. While Android more recently started to support checking permissions on first use, thereby creating context and transparency for the end-user, developers still need to actively opt-in to enable this functionality. In its current state, Android’s permission system tends to be impractical in many cases and can furthermore not be extended easily. In this dissertation, we have presented a line of work that addresses these shortcomings by retrofitting privacy controls to stock Android devices.

To provide users with an effective and practical means to protect their privacy, we focused on systems for security policy enforcement that could be deployed to stock Android devices, merely by downloading and installing an app. A particular challenge posed by Android’s security model was its app isolation paradigm, which actively prevents third-party apps from enforcing custom security policies. Currently, to deploy new or custom security extensions on their devices, users have to resort to specialized aftermarket firmware or need to gain root access, which poses a technological barrier for the majority of them.

Our first approach to establish security policy enforcement on the Android application layer used static on-device program rewriting to inline the reference monitor into the untrusted application. This technique allowed us to sidestep the app isolation mechanism by packaging both the reference monitor and the untrusted app into a single Android application. Security policies were enforced by rewriting all call sites to security-relevant methods in the untrusted app to divert control flow to the reference monitor. We implemented our approach as a regular Android app called APPGUARD, which was able to enforce complex user-defined security policies with negligible space and runtime overhead. Since its public release, APPGUARD has been downloaded more than one million times and has received the *5th German IT Security Award*, which demonstrates the need for practical privacy-preserving tools for mobile devices.

Despite this success, APPGUARD still had some shortcomings: First, instrumentation was performed statically. Therefore, changing the security policy at runtime required either over-instrumenting the program to account for all possible configuration or re-installing the app after every change, which is too cumbersome in practice. Furthermore, under certain circumstances, caller-side instrumentation could be incomplete, e.g., in scenarios where invocations of security-relevant methods could not be found by static analysis. Another shortcoming of APPGUARD was that it provided only insufficient app sandboxing functionality. Because the reference monitor and the untrusted application under observation shared the same process, they lacked the secure isolation necessary to ensure non-bypassability of the reference monitor and protection against tampering. Moreover, the modification and re-signing of applications required by inlining reference monitors violates Android’s signature-based same-origin model, which impedes automatic

app updates and user data migration. We were able to address some of these limitations in our paper on dynamic method hook injection that presented a technique for callee-side program instrumentation.

In this work, we proposed a novel approach for inline reference monitoring that abstained from static code rewriting entirely, which we achieved by diverting control flow in the Dalvik virtual machine. This approach was based on the insight that the VM-internal data structures that represent application code and libraries in memory are modifiable. Therefore, it was possible to alter the control flow by modifying the reference to a method's bytecode, thereby rerouting a call to this method to the reference monitor. This method is similar in spirit to changing function pointers and effectively enables callee-site instrumentation. Our initial empirical evaluation demonstrates that this approach incurs minimal runtime overhead. Using this technique, we were able to allow for dynamic security policy updates and reliable control flow diversion.

While we were able to address the issues above successfully, other problems persisted: In order to bootstrap the instrumentation of an untrusted app, the technique mentioned above relied on write access to the VM's memory. Without root privileges, such write access could only be obtained by statically injecting the bootstrapping code in the monitored app, though, thereby still necessitating re-signing of the application package. Another problem that remained unsolved was the issue of the reference monitor not being tamper-proof.

To address these shortcomings, we finally presented a system based on application virtualization and process-based privilege separation to securely encapsulate untrusted apps in a restricted execution environment within the context of another, trusted sandbox application. We were able to establish a restricted execution environment by leveraging Android's *isolated process* feature, which allows apps to completely de-privilege selected components. We introduced a novel app virtualization environment that proxies all system call and Binder channels of isolated apps. Technically, we leveraged our existing approach for dynamic method hook injection while simultaneously introducing new techniques, such as Binder IPC redirection through ServiceManager hooking. We realized our concept as a regular app named BOXIFY that could be deployed without firmware modifications or root privileges. A subsequent systematic evaluation of BOXIFY demonstrated its capability to enforce established and new security policies without incurring a significant runtime performance overhead.

To find a solution that provides users with a practical means to protect their privacy, we set out to enable customizations of Android's rigid permission system. We started by establishing fundamental policy enforcement on third-party apps using inlined reference monitors, as presented in Chapter 3. Faced with several limitations inherent to this approach, we then refined it by introducing a novel technique for dynamic method hook injection on Android's Java VM (Chapter 4). Finally, we introduced a system that leverages process-based privilege separation to provide a fully virtualized application environment that supports the enforcement of complex security policies on stock Android

devices (Chapter 5). The systems developed in this dissertation have not only been peer-reviewed but have proven themselves in real-world scenarios where actual users were able to address their privacy concerns.

Future Research Directions

We see the potential for future research to continue this line of work. With BOXIFY, we have introduced a generic framework to enforce arbitrary security policies within a virtual Android instance running on a stock Android device. Beyond the immediate privacy benefits for the end-user presented in this dissertation, BOXIFY offers all the security advantages of traditional sandboxing techniques and is thus of independent interest for future Android security research. We presented one use case in [S3] that leveraged BOXIFY for in-app ad-blocking that provided a high level of effectiveness in blocking ads while at the time being favorable for end-user deployment by abstaining from modifications of the operating system or any elevated privileges. Beyond this example, we envision a wide range of application domains for BOXIFY, such as application-layer taint-tracking for sandboxed apps [32], programmable security APIs in the spirit of ASM [51]/ASF [S1] to facilitate the extensibility of BOXIFY, as well as BOXIFY-based malware analysis tools.

Further, we envision improvements concerning Android device compatibility: The core issue here is that there is no single Android version, but a plethora of different releases and vendor-specific flavors. Any application layer solution – ours not being an exception – needs to account for the differences and specifics of each of these versions and flavors if they deviate from the open-source platform. This is particularly relevant if the solution relies on portions of the Android software stack that are not considered public and not intended for use by application developers as these parts can change at the manufacturer’s discretion. Whereas APPGUARD was largely robust against these specifics as it relied only on public APIs, in the case of BOXIFY, the various Android versions needed to be explicitly considered since it made extensive use of internal APIs. APPGUARD used static code analysis across different Android versions to generate a compatibility layer between the Android system and the sandbox, which needed to be updated for every new Android release. While this process was automatic, we envision a dynamic rule-based approach that is agnostic to the specific Android version used. This could be achieved by transforming Android Binder IPC communication based on a generic set of rules to attain transparent app sandboxing, instead of modeling every single Android middleware API endpoint. This way, only a few selected Android APIs would need explicit modeling and implementation.

Bibliography

Author's Papers for this Thesis

- [P1] 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 2013)*. Springer-Verlag, 2013.
- [P2] BACKES, M., GERLING, S., HAMMER, C., MAFFEI, M., and STYP-REKOWSKY, P. von. AppGuard – Fine-grained Policy Enforcement for Untrusted Android Applications. In: *Proceedings of the 8th International Workshop on Data Privacy Management (DPM 2013)*. Springer-Verlag, 2013.
- [P3] 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.
- [P4] STYP-REKOWSKY, P. von, GERLING, S., BACKES, M., and HAMMER, C. Callee-site Rewriting of Sealed System Libraries. In: *Proceedings of the 5th International Symposium on Engineering Secure Software and Systems (ESSoS 2013)*. Springer-Verlag, 2013.

Other Papers of the Author

- [S1] 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.
- [S2] BACKES, M., BUGIEL, S., SCHRANZ, O., STYP-REKOWSKY, P. von, and WEISSGERBER, S. ARTist: The Android runtime instrumentation and security toolkit. In: *Proceedings of the 2nd IEEE European Symposium on Security and Privacy (EuroS&P 2017)*. IEEE, 2017.
- [S3] BACKES, M., BUGIEL, S., STYP-REKOWSKY, P. von, and WISSFELD, M. Seamless In-App Ad Blocking on Stock Android. In: *Proceedings of the 2017 Mobile Security Technologies Workshop (MoST 2017)*. IEEE, 2017.

BIBLIOGRAPHY

- [S4] JAMROZIK, K., STYP-REKOWSKY, P. von, and ZELLER, A. Mining Sandboxes. In: *Proceedings of the 38th International Conference on Software Engineering (ICSE 2016)*. ACM, 2016.

Technical Reports of the Author

- [T1] BACKES, M., GERLING, S., HAMMER, C., MAFFEI, M., and STYP-REKOWSKY, P. von. *AppGuard – Real-time policy enforcement for third-party applications*. Tech. rep. A/02/2012. Saarland University, 2012.
- [T2] BACKES, M., GERLING, S., HAMMER, C., MAFFEI, M., and STYP-REKOWSKY, P. von. *AppGuard – Fine-grained Policy Enforcement for Untrusted Android Applications*. Tech. rep. A/02/2013. Saarland University, 2013.
- [T3] 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, 2014.

Other references

- [1] *Android Developer Dashboard*. <https://developer.android.com/about/dashboards/>. Last visited: 02/09/19.
- [2] *Android Developer's Guide*. <https://developer.android.com/guide/>. Last visited: 02/09/19.
- [3] 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.
- [4] BACKES, M., GERLING, S., and STYP-REKOWSKY, P. von. A Local Cross-Site Scripting Attack against Android Phones (2011). URL: <https://publications.cispa.saarland/36/>.
- [5] BACKES, M., BUGIEL, S., DERR, E., MCDANIEL, P., OCTEAU, D., and WEISGERBER, S. On Demystifying the Android Application Framework: Re-visiting Android Permission Specification Analysis. In: *Proceedings of the 25th USENIX Security Symposium (SEC 2016)*. USENIX Association, 2016.
- [6] BARRERA, D., KAYACIK, H. G., OORSCHOT, P. C. van, and SOMAYAJI, A. A Methodology for Empirical Analysis of Permission-Based Security Models and its Application to Android. In: *Proceedings of the 17th ACM Conference on Computer and Communication Security (CCS 2010)*. ACM, 2010.
- [7] BAUER, L., LIGATTI, J., and WALKER, D. *A Language and System for Composing Security Policies*. Tech. rep. TR-699-04. Princeton University, 2004.

-
- [8] BAUER, L., LIGATTI, J., and WALKER, D. Composing security policies with polymer. In: *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI 2005)*. ACM, 2005.
 - [9] BIANCHI, A., FRATANTONIO, Y., KRUEGEL, C., and VIGNA, G. NJAS: Sandboxing Unmodified Applications in Non-rooted Devices Running Stock Android. In: *Proceedings of the 5th ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM 2015)*. ACM, 2015.
 - [10] 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.
 - [11] 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.
 - [12] 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.
 - [13] 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 2011)*. USENIX Association, 2011.
 - [14] CHANDRASHEKHARA, S., MARCUS, K., SUBRAMANYA, R. G. M., KARVE, H. S., DANTU, K., and KO, S. Y. Enabling Automated, Rich, and Versatile Data Management for Android Apps with Bluemountain. In: *Proceedings of the 7th USENIX Conference on Hot Topics in Storage and File Systems (HotStorage 2015)*. USENIX Association, 2015.
 - [15] CHANDRASHEKHARA, S., KI, T., DANTU, K., and KO, S. Y. Duvel: Enabling Context-driven, Multi-profile Apps on Android Through Storage Sandboxing. In: *Proceedings of the 1st International Workshop on Edge Systems, Analytics and Networking (EdgeSys 2018)*. ACM, 2018.
 - [16] CHAUDHURI, A. Language-Based Security on Android. In: *Proceedings of the 4th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS 2009)*. ACM, 2009.
 - [17] CHAUDHURI, A., FUCHS, A., and FOSTER, J. *SCanDroid: Automated Security Certification of Android Applications*. Tech. rep. CS-TR-4991. University of Maryland, 2009. URL: <http://www.cs.umd.edu/~avik/papers/scandroidascaa.pdf>.

BIBLIOGRAPHY

- [18] CHEN, B. X. and BILTON, N. *Et Tu, Google? Android Apps Can Also Secretly Copy Photos*. 2012. URL: <http://bits.blogs.nytimes.com/2012/03/01/android-photos/>.
- [19] CHEN, F. and ROŞU, G. Java-MOP: A Monitoring Oriented Programming Environment for Java. In: *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)*. Springer-Verlag, 2005.
- [20] CHEN, Q. A., QIAN, Z., and MAO, Z. M. Peeking into Your App without Actually Seeing It: UI State Inference and Novel Android Attacks. In: *Proceedings of the 23rd USENIX Security Symposium (SEC 2014)*. USENIX Association, 2014.
- [21] *Chromium: Linux Sandboxing*. https://chromium.googlesource.com/chromium/src/+master/docs/linux_sandboxing.md. Last visited: 02/09/19.
- [22] CITRIX. *XenApp*. <https://www.citrix.com/products/xenapp-xendesktop/application-virtualization.html>. Last visited: 02/09/19.
- [23] 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.
- [24] DAN, M., JACOBS, B., LUNDBLAD, A., and PIESSENS, F. Security Monitor Inlining for Multithreaded Java. In: *Proceedings of the 23rd European Conference on Object-Oriented Programming (ECOOP 2009)*. 2009.
- [25] 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.
- [26] 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.
- [27] 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.
- [28] DESMET, L., JOOSEN, W., MASSACCI, F., NALIUKA, K., PHILIPPAERTS, P., PIESSENS, F., and VANOVERBERGHE, D. The S3MS.NET Run Time Monitor. *Electron. Notes Theor. Comput. Sci.* 253, 5 (Dec. 2009).
- [29] 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.

-
- [30] 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.
 - [31] ENCK, W., ONGTANG, M., and MCDANIEL, P. Understanding Android Security. *IEEE Security and Privacy* 7, 1 (2009).
 - [32] 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.
 - [33] 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.
 - [34] ERLINGSSON, Ú. The Inlined Reference Monitor Approach to Security Policy Enforcement. PhD thesis. Cornell University, 2004.
 - [35] 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.
 - [36] ERLINGSSON, U. and SCHNEIDER, F. B. SASI enforcement of security policies: a retrospective. In: *Proceedings of the 1999 Workshop on New Security Paradigms (NSPW 1999)*. 2000.
 - [37] FAHL, S., HARBACH, M., MUDERS, T., SMITH, M., BAUMGÄRTNER, L., and FREISLEBEN, B. Why Eve and Mallory love Android: An analysis of Android SSL (in) security. In: *Proceedings of the 19th ACM Conference on Computer and Communication Security (CCS 2012)*. ACM, 2012.
 - [38] FELT, A. P., GREENWOOD, K., and WAGNER, D. The Effectiveness of Application Permissions. In: *Proceedings of the 2nd Usenix Conference on Web Application Development (WebApps 2011)*. 2011.
 - [39] FRAGKAKI, E., BAUER, L., JIA, L., and SWASEY, D. Modeling and Enhancing Android's Permission System. In: *Proceedings of the 17th European Symposium on Research in Computer Security (ESORICS 2012)*. Springer-Verlag, 2012.
 - [40] GIBLER, C., CRUSSEL, J., ERICKSON, J., and CHEN, H. *AndroidLeaks: Detecting Privacy Leaks in Android Applications*. Tech. rep. CSE-2011-10. University of California Davis, 2011.
 - [41] 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.

BIBLIOGRAPHY

- [42] GOLDBERG, I., WAGNER, D., THOMAS, R., and BREWER, E. A. A Secure Environment for Untrusted Helper Applications Confining the Wily Hacker. In: *Proceedings of the 6th Conference on USENIX Security Symposium, Focusing on Applications of Cryptography (SSYM 1996)*. USENIX Association, 1996.
- [43] 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.
- [44] GRACE, M. C., ZHOU, Y., WANG, Z., and JIANG, X. Systematic Detection of Capability Leaks in Stock Android Smartphones. In: *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS 2012)*. The Internet Society, 2012.
- [45] GRUVER, B. *Smali: A assembler/disassembler for Android's dex format*. <https://github.com/JesusFreke/smali>. Last visited: 02/09/19.
- [46] GUO, P. J. and ENGLER, D. CDE: Using System Call Interposition to Automatically Create Portable Software Packages. In: *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference (USENIXATC 2011)*. USENIX Association, 2011.
- [47] HACKBORN, D. *Android Developer Group: Advantage of introducing Isolated-process tag within Services in JellyBean*. <https://groups.google.com/forum/?fromgroups=#!topic/android-developers/pk45eUFmKcM>. Last visited: 02/10/19. 2012.
- [48] HAMLEN, K. W. and JONES, M. Aspect-oriented in-lined reference monitors. In: *Proceedings of the 3rd ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS 2008)*. 2008.
- [49] HAMLEN, K. W., MORRISSETT, G., and SCHNEIDER, F. B. Certified In-lined Reference Monitoring on .NET. In: *Proceedings of the 1st ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS 2006)*. 2006.
- [50] 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.
- [51] 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.
- [52] 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.

-
- [53] *Java SE Documentation: Security Specification*. <https://docs.oracle.com/javase/7/docs/technotes/guides/security/spec/security-specTOC.fm.html>. Last visited: 02/09/19.
 - [54] 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.
 - [55] 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.
 - [56] 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.
 - [57] LIU, B., LIU, B., JIN, H., and GOVINDAN, R. Efficient Privilege De-Escalation for Ad Libraries in Mobile Apps. In: *Proceedings of the 13th International Conference on Mobile Systems, Applications, and Services (MobiSys 2015)*. ACM, 2015.
 - [58] MENG, W., DING, R., CHUNG, S. P., HAN, S., and LEE, W. The Price of Free: Privacy Leakage in Personalized Mobile In-Apps Ads. In: *Proceedings of the 23rd Annual Network and Distributed System Security Symposium (NDSS 2016)*. The Internet Society, 2016.
 - [59] MICROSOFT. *Application Virtualization (App-V)*. <https://docs.microsoft.com/en-us/windows/application-management/app-v/appv-for-windows>. Last visited: 02/09/19.
 - [60] MOULU, A. *Android OEM's applications (in)security and backdoors without permission*. https://www.sh4ka.fr/Android_OEM_applications_insecurity_and_backdoors_without_permission.pdf. Last visited: 02/09/19.
 - [61] MULLINER, C., OBERHEIDE, J., ROBERTSON, W., and KIRDA, E. PatchDroid: Scalable Third-party Security Patches for Android Devices. In: *Proceedings of the 29th Annual Computer Security Applications Conference (ACSAC 2013)*. ACM, 2013.
 - [62] 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.
 - [63] ONGTANG, M., BUTLER, K. R. B., and MCDANIEL, P. D. Porscha: policy oriented secure content handling in Android. In: *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC 2010)*. ACM, 2010.

BIBLIOGRAPHY

- [64] 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.
- [65] 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.
- [66] PORTER FELT, A., WANG, H. J., HANNA, A. M. and Steve, and CHIN, E. Permission Re-Delegation: Attacks and Defenses. In: *Proceedings of the 20th Usenix Security Symposium (SEC 2011)*. USENIX Association, 2011.
- [67] 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.
- [68] PROVOS, N. Improving Host Security with System Call Policies. In: *Proceedings of the 12th Usenix Security Symposium (SEC 2003)*. USENIX Association, 2003.
- [69] PROVOS, N., FRIEDL, M., and HONEYMAN, P. Preventing Privilege Escalation. In: *Proceedings of the 12th Usenix Security Symposium (SEC 2003)*. USENIX Association, 2003.
- [70] RASTHOFER, S., ARZT, S., LOVAT, E., and BODDEN, E. DroidForce: Enforcing Complex, Data-Centric, System-Wide Policies in Android. In: *Proc. 9th International Conference on Availability, Reliability and Security (ARES'14)*. IEEE, 2014.
- [71] RASTOGI, V., CHEN, Y., and JIANG, X. DroidChameleon: Evaluating Android Anti-malware Against Transformation Attacks. In: *Proceedings of the 8th ACM Symposium on Information, Computer and Communication Security (ASIACCS 2013)*. ACM, 2013.
- [72] RAVAL, N., RAZEEN, A., MACHANAVAJJHALA, A., COX, L. P., and WARFIELD, A. Permissions Plugins As Android Apps. In: *Proceedings of the 17th International Conference on Mobile Systems, Applications, and Services (MobiSys 2019)*. ACM, 2019.
- [73] 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.
- [74] RUSSELLO, G., JIMENEZ, A. B., NADERI, H., and MARK, W. van der. FireDroid: Hardening Security in Almost-stock Android. In: *Proceedings of the 29th Annual Computer Security Applications Conference (ACSAC 2013)*. ACM, 2013.
- [75] SAMSUNG ELECTRONICS. *Whitepaper: Samsung Knox Security Solution*. <https://www.samsungknox.com/docs/SamsungKnoxSecuritySolution.pdf>. Last visited: 02/09/19. 2013.

-
- [76] SARNO, D. *Twitter stores full iPhone contact list for 18 months, after scan*. <http://articles.latimes.com/2012/feb/14/business/la-fi-tn-twitter-contacts-20120214>. Last visited: 02/10/19. 2012.
 - [77] 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.
 - [78] SCHNEIDER, F. B. Enforceable Security Policies. *ACM Transactions on Information and System Security* 3, 1 (2000).
 - [79] STATISTA. *Global mobile OS market share in sales to end users from 1st quarter 2009 to 2nd quarter 2018*. <https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/>. Last visited: 02/07/19. 2019.
 - [80] STATISTA. *Number of available applications in the Google Play Store from December 2009 to June 2019*. <https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/>. Last visited: 02/07/19. 2019.
 - [81] *The Chromium Projects: OSX Sandboxing Design*. <https://dev.chromium.org/developers/design-documents/sandbox/osx-sandboxing-design>. Last visited: 02/09/19.
 - [82] *The Chromium Projects: Sandbox (Windows)*. <https://chromium.googlesource.com/chromium/src/+master/docs/design/sandbox.md>. Last visited: 02/09/19.
 - [83] 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.
 - [84] 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.
 - [85] *Wine: Run Windows applications on Linux, BSD, Solaris and Mac OS X*. <https://www.winehq.org>. Last visited: 02/09/19.
 - [86] 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.

BIBLIOGRAPHY

- [87] WU, D., CHENG, Y., GAO, D., LI, Y., and DENG, R. H. SCLib: A Practical and Lightweight Defense Against Component Hijacking in Android Applications. In: *Proceedings of the 8th ACM Conference on Data and Application Security and Privacy (CODASPY 2018)*. ACM, 2018.
- [88] 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.
- [89] 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.
- [90] XUAN, C., CHEN, G., and STUNTEBECK, E. DroidPill: Pwn Your Daily-Use Apps. In: *Proceedings of the 12th ACM Asia Conference on Computer and Communications Security (ASIACCS 2017)*. ACM, 2017.
- [91] XUE, L., ZHOU, Y., CHEN, T., LUO, X., and GU, G. Malton: Towards On-device Non-invasive Mobile Malware Analysis for ART. In: *Proceedings of the 26th USENIX Security Symposium (SEC 2017)*. USENIX Association, 2017.
- [92] YEE, B., SEHR, D., DARDYK, G., CHEN, J. B., MUTH, R., ORMANDY, T., OKASAKA, S., NARULA, N., and FULLAGAR, N. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In: *Proceedings of the 30th IEEE Symposium on Security and Privacy (Oakland 2009)*. IEEE, 2009.
- [93] YOU, W., LIANG, B., SHI, W., ZHU, S., WANG, P., XIE, S., and ZHANG, X. Reference Hijacking: Patching, Protecting and Analyzing on Unmodified and Non-rooted Android Devices. In: *Proceedings of the 38th International Conference on Software Engineering (ICSE 2016)*. ACM, 2016.
- [94] *Zero-Permission Android Applications*. <https://www.leviathansecurity.com/blog/zero-permission-android-applications/>. Last visited: 02/09/19.
- [95] *Zero-Permission Android Applications (Part 2)*. <https://www.leviathansecurity.com/blog/zero-permission-android-applications-part-2/>. Last visited: 02/09/19.
- [96] 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.
- [97] 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.

- [98] 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.